FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Eine Entwicklungsumgebung für iterative Eigenwertverfahren in MATLAB

Britta Janssen

FZJ-ZAM-IB-2005-18

Dezember 2005

(letzte Änderung: 15.12.2005)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Die Berechnung von Eigenwerten und Eigenvektoren großer, dünn besetzter Matrizen ist in den Natur- und Ingenieurwissenschaften eine häufig wiederkehrende Aufgabe. Meistens benötigt man in der Praxis nur einige Eigenwerte, z.B. die kleinsten. Es wäre sehr ineffektiv, diese mit den gängigen Verfahren wie z.B. einer QR-Zerlegung zu bestimmen, da eine QR-Zerlegung alle Eigenwerte und Eigenvektoren einer $n \times n$ Matrix berechnet und einen Aufwand der Ordnung $\mathcal{O}(n^3)$ hat. Hier sind Projektionsverfahren wesentlich effizienter. Ein Beispiel für ein Projektionsverfahren ist das Jacobi-Davidson-Verfahren. Hierbei wird das Eigenwertproblem in einen geeigneten niederdimensionalen Unterraum projiziert, in dem Eigenwerte und Eigenvektoren der projizierten Matrix berechnet werden. Diese sind Ritzwerte bzw. Ritzvektoren der ursprünglichen Matrix, d.h. Approximationen der Eigenwerte bzw. Eigenvektoren. Sind diese Näherungen noch nicht genau genug, so wird eine Korrekturgleichung gelöst. Die Lösung liefert einen Vektor, mit dem der Unterraum erweitert werden kann. Diese Prozedur wird so lange fortgesetzt, bis die Näherungen für Eigenwerte und Eigenvektoren einer vorgegebenen Genauigkeit genügen.

Im Rahmen dieser Diplomarbeit ist eine Entwicklungsumgebung für iterative Eigenwertverfahren in MATLAB entstanden, die es erleichtert, verschiedene Algorithmen zur Berechnung von Eigenwerten und Eigenvektoren zu testen. Die implementierte graphische Benutzeroberfläche erlaubt zur Zeit die Untersuchung der Verfahren von Arnoldi und Lanczos sowie des Jacobi-Davidson-Verfahrens. Beim Jacobi-Davidson-Verfahren sind verschiedenste Kombinationsmöglichkeiten vorhanden. Matrizen können in drei verschiedenen Formaten in MATLAB eingelesen oder extern abgespeichert werden. Zur Auslagerung einiger Programmteile stehen Schnittstellen zur Verfügung.

Abstract

In various areas of Computational Science and Engineering (CSE) it is often required to compute eigenvalues and eigenvectors of large sparse matrices. Usually only some eigenvalues are needed, for example the minimal ones. It would be ineffective to calculate these with an algorithm like the QR-Iteration because the QR-Iteration calculates all eigenvalues and eigenvectors of a $n \times n$ matrix and the algorithm is of complexity $\mathcal{O}(n^3)$. In this case methods which project the matrix into lower dimensional subspaces are more efficient. One example of such a method is the Jacobi-Davidson algorithm. The eigenvalue problem is projected into a lower dimensional subspace and the eigenvalues und eigenvectors of the projected matrix are computed. These are ritz values and ritz vectors of the original matrix that means approximations for the eigenvalues and eigenvectors. If these approximations are not accurate enough, a correction equation is solved. The solution of the correction equation provides a vector to expand the subspace. This procedure continues until the approximations for the eigenvalues and eigenvectors satisfy a given tolerance.

In the diploma thesis an interactive, userfriendly environment for iterative eigenvalue methods in MATLAB is developed. Several algorithms for computing eigenvalues and eigenvectors can be tested. The implemented graphical user interface allows the analysis of the methods of Arnoldi, Lanczos and several combinations of the Jacobi-Davidson method. Matrices can be imported into MATLAB using three formats or can be stored externally. Program interfaces are available to connect to user written code.

Kapitel 1

Einleitung

Die Berechnung von Eigenwerten und Eigenvektoren von großen, dünn besetzten Matrizen ist in vielen Bereichen notwendig. In der Statik, einem Bereich der Mechanik, ist man beispielsweise an den Eigenschwingungen eines Bauteils interessiert. Bei Bauwerken wie Brücken benötigt man Eigenwerte, um diese so zu bauen, dass sie auch starken Umwelteinflüssen wie Erdbeben oder langandauernden Windströmungen standhalten (siehe hierzu [10]). Ein weiterer Bereich, in dem Eigenwerte berechnet werden müssen, ist die Quantenchemie. Hier beschreiben Eigenwerte zum Beispiel Energie- und Schwingungszustände von Atomen oder Molekülen (siehe auch [14]).

Meistens benötigt man allerdings nicht alle Eigenwerte einer Matrix, sondern nur einige wenige. Bei manchen Anwendungen sind nur die betragsgrößten Eigenwerte, bei anderen nur die betragskleinsten Eigenwerte von Belang. Auch kommt es vor, dass man an Eigenwerten um einen speziellen Wert interessiert ist. Es wäre in diesen Fällen äußerst ineffektiv, alle Eigenwerte der Matrix mit den gängigen Methoden, zum Beispiel einer QR-Zerlegung, zu berechnen, da der Aufwand viel zu hoch ist.

Will man nur wenige Eigenwerte einer großen Matrix berechnen, so ist das Jacobi-Davidson-Verfahren wesentlich effizienter. Beim Jacobi-Davidson-Verfahren wird die Matrix in einen niederdimensionalen Unterraum projiziert. Dort werden dann Eigenvektornäherungen bestimmt, eine Korrekturgleichung gelöst und der Unterraum entsprechend erweitert. So können nach und nach alle benötigten Eigenwerte und -vektoren einer Matrix bestimmt werden.
Ist man jedoch an allen Eigenwerten interessiert, so sollte man andere Methoden, zum Beispiel die QR-Zerlegung, nutzen.

Ziel dieser Arbeit ist es, eine Testumgebung für iterative Eigenwertverfahren in MATLAB zu entwickeln. Die Testumgebung stellt verschiedene Algorithmen zur Eigenwertberechnung bereit, so dass der Benutzer prüfen kann, welche Verfahren am effizientesten sind und die besten Ergebnisse für seine Problemstellung liefern. Hier soll mit kleinen Matrizen gleicher Struktur getestet werden, wie schnell und wie exakt welches Verfahren die gewünschten Eigenwerte und -vektoren bestimmt. Danach können die Eigenwerte der zu Grunde liegenden größeren Matrix mittels dieses Verfahrens berechnet werden.
Die Benutzeroberfläche soll leicht und ohne genaue Kenntnis der Syntax von MATLAB bedienbar sein. Außerdem soll die Auslagerung von bestimmten Programmteilen, z.B. die Lösung der Korrekturgleichung, ermöglicht werden.

Das Kapitel 2 beinhaltet wichtige mathematische Grundlagen. Hier wird unter anderem erklärt, worin der Unterschied zwischen Eigen- und Ritzwerten besteht.

Kapitel 3 geht auf grundlegende Algorithmen, zum Beispiel das Gram-Schmidt- oder das CG-Verfahren, ein.

Das Kapitel 4 beschreibt den mathematischen Hintergrund des Jacobi-Davidson-Verfahrens. Dazu werden zunächst das Jacobi- und das Davidson-Verfahren erklärt, da das Jacobi-Davidson-Verfahren aus beiden Verfahren hervorgegangen ist. Weiterhin wird das Jacobi-Davidson-Verfahren sowohl mit Standard-Ritzwerten als auch mit harmonischen Ritzwerten erklärt.
Das Verfahren mit Standard-Ritzwerten konvergiert gewöhnlich gegen äußere Eigenwerte, wohingegen harmonische Ritzwerte gegen innere Eigenwerte konvergieren.

Kapitel 2 bis 4 entstanden in Zusammenarbeit mit Herrn Rene Puttin, der zur gleichen Zeit am Zentralinstitut für Angewandte Mathematik eine Diplomarbeit zum Thema Jacobi-Davidson-Verfahren geschrieben hat. Seine Aufgabe war die Optimierung und Parallelisierung eines bereits bestehenden Programms in Fortran.

Das Kapitel 5 beschäftigt sich mit den Programmiertechniken in MATLAB. Es wird erläutert, wie Benutzeroberflächen mit Hilfe von ***guide*** (**G**raphical **U**ser **I**nterface **D**evelopment **E**nvironment) erstellt werden können. Außerdem wird erklärt, wie man externe, in C oder Fortran geschriebene Funktionen in MATLAB einbindet.

Das Kapitel 6 beschreibt die Funktionalität des Programmpaketes im Einzelnen. Es werden sowohl der Aufbau der Benutzeroberfläche als auch die Schnittstellen zur externen Berechnung erläutert.

Kapitel 2

Mathematische Grundlagen

Nach einer kurzen Einleitung, in der auch die praktische Bedeutung der Eigenwerte erklärt wurde, werden nun im zweiten Kapitel die benötigten mathematischen Definitionen eingeführt. Dazu wird eine kurze Zusammenfassung der, im Hinblick auf diese Arbeit, wichtigen Begriffe der Linearen Algebra gegeben. Die Definitionen und Bezeichnungen werden für reelle Zahlen und Systeme, die aus reellen Zahlen bestehen, eingeführt. Wenn nichts anderes erwähnt ist, gelten sie aber auch analog für komplexe Zahlen. Aufbauend auf diesen Definitionen werden dann im nächsten Kapitel die numerischen Verfahren zur Bestimmung der Eigenwerte erläutert.

2.1 Matrizen und Vektoren

Wie im ersten Kapitel dargestellt, laufen viele Probleme auf lineare Gleichungssysteme

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & \vdots & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array}$$

hinaus. Diese lassen sich einfacher in Form von Matrizen und Vektoren darstellen. Dazu wird im Folgenden erklärt, was man unter Vektoren beziehungsweise Matrizen versteht.

Bezeichnung 1. *Sei x ein m-Tupel von reellen Zahlen*

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

dann bezeichnet man x als m-dimensionalen Spaltenvektor. Man schreibt auch: $x \in \mathbb{R}^m$.

Bezeichnung 2. *Sei A ein zweidimensionales Anordnungssystem, bestehend aus Zeilen und Spalten von reellen Werten,*

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

dann bezeichnet man A als $m \times n$-dimensionale reelle Matrix. Man schreibt auch: $A \in \mathbb{R}^{m \times n}$.

Analog zu den Skalaren lassen sich auch für Matrizen und Vektoren Addition und Multiplikation definieren.

Bemerkung 1. *Die Addition von Vektoren und Matrizen erfolgt komponentenweise.*

Die Multiplikation von Vektoren ist durch das Skalarprodukt gegeben.

Definition 1. *Seien $x, y \in \mathbb{R}^m$ Vektoren, dann nennt man*

$$< x, y >= \sum_{i=1}^{m} x_i y_i$$

Skalarprodukt der Vektoren x und y.

Im komplexen Zahlenbereich ist das Skalarprodukt mit Hilfe der komplex konjugierten Komponente des ersten Vektors definiert.

Bezeichnung 3. *Seien $x, y \in \mathbb{R}^m$ und $z = x + iy, z \in \mathbb{C}^m$, dann nennt man*

$$\overline{z} = x - iy$$

den komplex konjugierten Vektor zum komplexen Vektor z.

Definition 2. *Seien $u, v \in \mathbb{C}^m$ komplexe Vektoren, dann heißt*

$$< u, v >= \sum_{i=1}^{m} \overline{u}_i v_i$$

Skalarprodukt der Vektoren u und v.

Bemerkung 2. *Im Folgenden wird oft auch die verkürzte Schreibweise $x^\star y$ für $< x, y >$ benutzt.*

Bemerkung 3. *Gilt für das Skalarprodukt zweier Vektoren x und y*

$$< x, y >= 0$$

so stehen die beiden Vektoren senkrecht aufeinander. Man sagt auch, die Vektoren sind orthogonal und schreibt:

$$x \perp y$$

Um Längen von Vektoren angeben zu können, führt man sogenannte Normen ein.

Definition 3. *Sei $x \in \mathbb{R}^m$ ein Vektor, dann heißt*

$$\|x\|_p = \left\{ \sum_{i=1}^{m} x_i^p \right\}^{1/p}$$

für $1 \leq p < \infty$ p-Norm des Vektors x.

Bemerkung 4. *Die 2er-Norm, auch euklidische Norm genannt,*

$$\|x\|_2 = \sqrt{< x, x >}$$

beschreibt die Länge des Vektors x im euklidischen Raum.

Häufig benötigt man auch die transponierte beziehungsweise komplex konjugiert transponierte Matrix.

Bezeichnung 4. *Sei $A \in \mathbb{R}^{m\times n}$ eine Matrix, dann nennt man*

$$A^T \text{ mit } a_{ij}^T = a_{ji} \quad \forall\, i = 1, ..., n,\ j = 1, ..., m$$

die transponierte Matrix zur Matrix A.

Bezeichnung 5. *Sei $A \in \mathbb{C}^{m\times n}$ eine komplexe Matrix, dann nennt man*

$$A^\star \text{ mit } a_{ij}^\star = \overline{a}_{ji} \quad \forall\, i = 1, ..., n,\ j = 1, ..., m$$

die komplex konjugiert transponierte Matrix zur Matrix A.

Damit lässt sich auch für Matrizen ein Produkt definieren.

Definition 4. *Seien $A \in \mathbb{R}^{m\times n}$ und $B \in \mathbb{R}^{n\times k}$ Matrizen, dann nennt man*

$$C = AB \text{ mit } c_{ij} =< (A^T)^{(i)}, B^{(j)} > \quad \forall\, i = 1, ..., m,\ j = 1, ..., k$$

Matrix-Matrix-Multiplikation der Matrizen A und B. Dabei steht $A^{(i)}$ für den i-ten Spaltenvektor der Matrix A.

Bemerkung 5. *In gleicher Weise lässt sich auch die Multiplikation einer $m \times n$-Matrix mit einem n-dimensionalen Vektor einführen. Der Vektor wird dabei als $n \times 1$-Matrix aufgefasst:*

$$y = Ax \text{ mit } y_i =< (A^T)^{(i)}, x > \quad \forall\, i = 1, ..., m$$

2.2 Lineare Gleichungssysteme

Mit Hilfe von Matrizen und Vektoren ist man in der Lage, ein lineares Gleichungssystem in einer einfachen und kompakten Form aufzuschreiben, die in umfangreichen Rechnungen deutlich leichter zu handhaben ist.

Bezeichnung 6. *Seien $A \in \mathbb{R}^{m\times n}$ eine Matrix und $x \in \mathbb{R}^n, b \in \mathbb{R}^m$ Vektoren, dann nennt man*

$$Ax = b$$

ein lineares Gleichungssystem.

Nicht nur für reelle Zahlen, sondern auch für Matrizen lassen sich inverse und neutrale Elemente definieren.

Bemerkung 6. *Die Nullmatrix*

$$O \in \mathbb{R}^{m\times n} \quad \text{mit } o_{ij} = 0 \quad \forall\, i = 1, ..., m,\ j = 1, ..., n$$

stellt das neutrale Element bezüglich der Matrixaddition dar.

Bemerkung 7. *Sei $A \in \mathbb{R}^{m\times n}$ eine Matrix, dann stellt die Matrix $-A$ das inverse Element von A bezüglich der Addition dar.*

Bemerkung 8. *Die Einheitsmatrix*

$$E \in \mathbb{R}^{n\times n} \quad \text{mit } e_{ij} = \delta_{ij} = \begin{cases} 1 \text{ für } i = j \\ 0 \text{ sonst} \end{cases}$$

stellt das neutrale Element bezüglich der Matrixmultiplikation dar. Häufig wird die Einheitsmatrix auch mit I bezeichnet.

Bemerkung 9. *Sei $A \in \mathbb{R}^{n\times n}$, dann stellt*

$$A^{-1} \text{ mit } AA^{-1} = A^{-1}A = E$$

das inverse Element der Matrix A bezüglich der Multiplikation dar. Im Allgemeinen nennt man A^{-1} die Inverse von A.

Bemerkung 10. *Es ist zu beachten, dass die Inverse nur für quadratische Matrizen $A \in \mathbb{C}^{n\times n}$ definiert ist und selbst dann nicht immer existiert.*

Im Folgenden soll nun genauer betrachtet werden, wann eine inverse Matrix existiert.

Definition 5. *Seien $x_i \in \mathbb{R}^m, i = 1, ..., n$ Vektoren, $x_i \neq 0$ und $\alpha_i \in \mathbb{R}, i = 1, ..., n$ reelle Zahlen, dann heißen die x_i linear unabhängig, falls gilt:*

$$\sum_{i=1}^{n} \alpha_i x_i = 0 \Leftrightarrow \alpha_i = 0 \quad \forall\, i = 1, ..., n$$

Bezeichnung 7. *Die Anzahl der linear unabhängigen Spaltenvektoren einer Matrix bezeichnet man als Spaltenrang. Analog dazu bezeichnet man die Anzahl der linear unabhängigen Zeilenvektoren als Zeilenrang. Es lässt sich feststellen, dass für jede Matrix der Spalten- und Zeilenrang identisch ist. Deshalb spricht man in der Regel nur vom Rang einer Matrix.*

Bezeichnung 8. *Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt A invertierbar, falls gilt:*

$$rang(A) = n$$

Bemerkung 11. *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix und $x, b \in \mathbb{R}^n$ Vektoren, dann hat das Gleichungssystem*

$$Ax = b$$

genau dann eine eindeutige Lösung, wenn A invertierbar ist.
Als eindeutige Lösung ergibt sich:

$$x = A^{-1}b$$

Löst man ein Gleichungssystem numerisch, so benötigt man eine Aussage darüber, wie genau die aktuelle Näherung an der exakten Lösung liegt. Ein Hinweis, wie gut diese Näherung ist, liefert das Residuum.

Bezeichnung 9. *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix und $x, \hat{x}, b \in \mathbb{R}^n$ Vektoren, wobei $\hat{x}$ eine Näherungslösung des Gleichungssystems*

$$Ax = b$$

darstellt, dann nennt man

$$r = b - A\hat{x}, \qquad r \in \mathbb{R}^n$$

das Residuum der Näherung.

2.3 Arten von Matrizen

Matrizen lassen sich aufgrund ihrer Eigenschaften in verschiedene Klassen einteilen.

Bezeichnung 10. *Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt A symmetrisch, falls gilt:*

$$A = A^T$$

Bezeichnung 11. *Sei $A \in \mathbb{C}^{n\times n}$ eine Matrix, dann heißt A hermitesch, falls gilt:*

$$A = A^\star$$

Bezeichnung 12. *Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt A anti- oder schiefsymmetrisch, falls gilt:*

$$A = -A^T$$

Bezeichnung 13. *Sei $A \in \mathbb{C}^{n\times n}$ eine komplexe Matrix, dann heißt A anti- oder schiefhermitesch, falls gilt:*

$$A = -A^\star$$

Bezeichnung 14. *Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt A positiv definit, falls gilt:*

$$< x, Ax > \; > 0 \qquad \forall\, x \in \mathbb{R}^n,\; x \neq 0$$

Bezeichnung 15. *Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt A diagonal dominant beziehungsweise streng diagonal dominant, falls gilt ([21]):*

$$|a_{i,i}| \geq \sum_{j\neq i} |a_{ij}| \qquad \forall\, i = 1, ..., n$$

beziehungsweise

$$|a_{i,i}| > \sum_{j\neq i} |a_{ij}| \qquad \forall\, i = 1, ..., n$$

Bezeichnung 16. *Sei $Q \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt Q orthogonal, falls gilt:*

$$QQ^T = E \quad \Leftrightarrow \quad Q^T = Q^{-1}$$

Bezeichnung 17. *Sei $Q \in \mathbb{C}^{n\times n}$ eine komplexe Matrix, dann heißt Q unitär, falls gilt:*

$$QQ^\star = E \quad \Leftrightarrow \quad Q^\star = Q^{-1}$$

Bemerkung 12. *Die Spaltenvektoren $q^{(i)}$ einer orthogonalen Matrix Q bilden ein Orthonormalsystem, das heißt:*

$$\|q^{(i)}\|_2 = 1 \quad \forall\, i = 1, ..., n$$

und

$$< q^{(i)}, q^{(j)} >= \delta_{ij} = \begin{cases} 1 & \text{für } i = j \\ 0 & \text{sonst} \end{cases} \qquad \forall\, i, j = 1, ..., n$$

Des Weiteren kann man Matrizen auch anhand ihrer Besetzungsstrukturen unterscheiden.

Bezeichnung 18. *Sei $A \in \mathbb{R}^{m\times n}$ eine Matrix, dann heißt A untere beziehungsweise obere Dreiecksmatrix, falls gilt:*

$$a_{ij} = 0 \quad \forall\, i < j \quad \textit{beziehungsweise} \quad a_{ij} = 0 \quad \forall\, i > j$$

Bemerkung 13. *Eine obere / untere Dreiecksmatrix mit genau einer zusätzlichen Diagonale unterhalb / oberhalb der Hauptdiagonalen bezeichnet man als Hessenbergmatrix, das heißt die Matrix ist in Hessenbergform.*

Bezeichnung 19. *Sei $A \in \mathbb{R}^{m\times n}$ eine Matrix, dann heißt A Bandmatrix mit n_l Subdiagonalen (Nebendiagonalen links/unterhalb der Hauptdiagonalen) und n_r Superdiagonalen (Nebendiagonalen rechts/oberhalb der Hauptdiagonalen), falls gilt:*

$$a_{ij} = 0 \quad \forall\, a_{ij} \text{ mit } i - j < n_l \ \vee \ j - i > n_r$$

Bemerkung 14. *Bandmatrizen mit jeweils einer Sub- und Superdiagonalen bezeichnet man als Tridiagonalmatrizen.*

Bezeichnung 20. *Neben den klassischen Besetzungsarten existieren auch willkürlich dünn besetzte Matrizen, die man als Sparse-Matrizen bezeichnet.*

Als weitere wichtige Matrix ist die sogenannte charakteristische Matrix zu nennen.

Bezeichnung 21. *Seien $A \in \mathbb{R}^{n\times n}$ eine quadratische Matrix und λ einer ihrer Eigenwerte (siehe Kapitel 2.4), dann nennt man die Matrix $A - \lambda I$ charakteristische Matrix.*

2.4 Eigen- und Ritzwerte

Anwendungen laufen in der Praxis meistens auf zwei grundlegende Problemstellungen hinaus. Die erste ist die Lösung von Gleichungssystemen, welche bereits im Kapitel 2.2 vorgestellt wurde. Das zweite Problem wird als sogenanntes Eigenwertproblem bezeichnet.

Definition 6. *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix, $x \in \mathbb{R}^n, x \neq 0$ ein Vektor und λ ein Skalar, die die Gleichung*

$$Ax = \lambda x$$

erfüllen, dann nennt man λ Eigenwert und x Eigenvektor der Matrix A. (λ, x) wird auch als Eigenpaar bezeichnet.

Bemerkung 15. *Die Eigenwerte von A entsprechen den Nullstellen des charakteristischen Polynoms*

$$\det(A - \lambda I) = 0$$

Es lässt sich direkt erkennen, dass für Eigenwertprobleme mit einer Matrixdimension größer als vier mit Hilfe des charakteristischen Polynoms im Allgemeinen keine analytische Lösung mehr gefunden werden kann, da für Gleichungen mit höherer Ordnung als vier keine geschlossene Lösungsformel existiert.

Auch für das Eigenwertproblem lässt sich ein Residuum definieren, mit dem man in numerischen Rechnungen Fehlerschranken aufstellen kann.

Bezeichnung 22. *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix, $\theta \in \mathbb{R}$ eine Näherung für den Eigenwert λ und $y \in \mathbb{R}^n, y \neq 0$ eine Näherung für den zugehörigen Eigenvektor x der Matrix A, dann bezeichnet man mit*

$$r = Ay - \theta y$$

das Residuum des genäherten Eigenpaares.

Bei der numerischen Bestimmung der Eigenwerte großer Matrizen werden häufig Unterraumtechniken verwendet.

Definition 7. *$U \subset \mathbb{R}^n$ heißt Unterraum des $\mathbb{R}^n$, falls gilt:*

- $U \neq \emptyset$
- $x + y \in U \quad \forall\, x, y \in U$ *(Additivität)*
- $\alpha \cdot x \in U \quad \forall\, x \in U, \;\; \alpha \in \mathbb{R}$ *(Homogenität)*

Ein Unterraum wird durch seine Basis festgelegt.

Definition 8. *Seien $U \subset \mathbb{R}^n$ ein Unterraum und $u_i \in \mathbb{R}^n, i = 1, ..., k, \;\; k < n$ Vektoren, dann heißt $\{u_1, ..., u_k\}$ Basis des Unterraums U, falls*

- *die Vektoren u_i linear unabhängig sind*
- *die lineare Hülle $L = \left\{\sum_{i=1}^{k} \alpha_i u_i, \alpha_i \in \mathbb{R}, i = 1, ..., k\right\} = U$ ist.*

Weiterhin bezeichnet man mit $dim(U) = k$ die Dimension des Unterraums.

Die meisten Unterraumverfahren arbeiten mit speziellen Unterräumen des $\mathbb{C}^n$, den sogenannten Krylov-Räumen.

Definition 9. *Sei $A \in \mathbb{C}^{n\times n}$ eine quadratische Matrix und $q \in \mathbb{C}^n$ ein Vektor, dann wird der Krylov-Raum von A über q definiert als*

$$K = K(A, q) = K_m(A, q) = \text{span } \{q, Aq, ..., A^{m-1}q\}.$$

Der Vektor q wird dabei als Startvektor der Krylov-Sequenz bezeichnet.

Bemerkung 16. *Multipliziert man die Systemmatrix A beliebig oft von links an einen Vektor q, der sich im Krylov-Raum befindet, so liegt das Ergebnis im nächstgrößeren Krylov-Raum beziehungsweise im gleichen Krylov-Raum, falls dieser A-invariant ist. Eine Erweiterung ist dann nicht mehr möglich.*

Bemerkung 17. *Seien $t \in \mathbb{C}^n$ ein Vektor und $U \subset \mathbb{C}^n$ ein Unterraum, und es gelte:*

$$t \perp u \qquad \forall\, u \in U$$

so schreibt man

$$t \perp U$$

und definiert

$$U^{\perp} = \{t \in \mathbb{C}^n : t \perp U\}$$

als orthogonales Komplement des Unterraums U.

Projiziert man eine Matrix A in einen niederdimensionalen Unterraum U und bestimmt die Eigenwerte der projizierten Matrix, so sind dies Ritzwerte der Matrix A im Bezug auf den Unterraum U.

Definition 10. *Seien $U \subset \mathbb{R}^n$ ein Unterraum, $A \in \mathbb{R}^{n\times n}$ eine Matrix, $z \in U, z \neq 0$ ein Vektor und $\theta \in \mathbb{C}$ ein Skalar, die die Ritz-Galerkin-Bedingung*

$$Az - \theta z \perp U$$

erfüllen, dann nennt man θ Ritzwert und z Ritzvektor zu A bezüglich des Unterraums U. (θ, z) wird auch als Ritzpaar bezeichnet.

Zur Bestimmung von Eigenwerten aus dem Innern des Spektrums werden harmonische Ritzwerte (siehe [9]) benötigt.

Definition 11. *Seien $U \subset \mathbb{R}^n$ ein Unterraum, $A \in \mathbb{R}^{n\times n}$ eine Matrix, $\theta \in \mathbb{C}$ ein Skalar, dann ist θ genau dann ein harmonischer Ritzwert von A bezüglich des Unterraums U, wenn θ^{-1} ein Ritzwert von A^{-1} in Bezug auf den Unterraum U ist.*

Kapitel 3

Grundlegende Algorithmen

In diesem Kapitel werden einige grundlegende mathematische Verfahren beschrieben, die später in den komplexeren Eigenwertverfahren verwendet werden.

3.1 Gram-Schmidt-Verfahren

Die Krylov-Unterraumverfahren, die im nächsten Kapitel vorgestellt werden, arbeiten häufig mit orthonormalen Basen für die Unterräume. Die Unterräume werden dabei sukzessiv erweitert, das heißt in jedem Iterationsschritt wird zu einer bestehenden Basis ein neuer Basisvektor hinzugefügt. Das Orthogonalisierungsverfahren von Gram-Schmidt liefert dazu einen passenden Algorithmus, der einen Vektor t bezüglich der orthonormalen Vektoren $v_i, i = 1, ..., m-1$ orthonormiert.

Abbildung 3.1: Orthogonalisierung von Gram-Schmidt

Da dieses Gram-Schmidt-Verfahren in einigen Fällen numerisch instabil ist, verwendet man häufig das modifizierte Gram-Schmidt-Verfahren.

Abbildung 3.2: Modifiziertes Gram-Schmidt-Verfahren

3.2 Arnoldi-/Lanczos-Verfahren

Im Folgenden werden nun kurz die Arnoldi- und die Lanczos-Methode vorgestellt, welche iterative Verfahren zur Lösung von Eigenwertproblemen sind.

Arnoldi-Verfahren

Der Arnoldi-Algorithmus dient als Projektionsmethode für $A \in \mathbb{R}^{n\times n}$, wobei dieser erst effizient ist, wenn n groß ist. Er liefert Orthonormalbasen von Krylov-Räumen. Den Krylov-Raum benutzt man dann als Suchraum für die Projektionsmethode.
Der Arnoldi-Algorithmus berechnet zu einer Matrix A eine unitäre Matrix $Q = [q_1, ..., q_k]$, so dass $Q^{(-1)}AQ = H$ Hessenbergform besitzt (siehe 2.3 auf Seite 9). H ist dabei die Galerkin-Projektion (siehe 2.4 auf Seite 11) von A in den Krylov-Raum. Berechnet man nun die Eigenpaare von H, so sind dies Ritzpaare von A bezüglich des aufgestellten Krylov-Raums K und somit Näherungen für ein Eigenpaar von A.

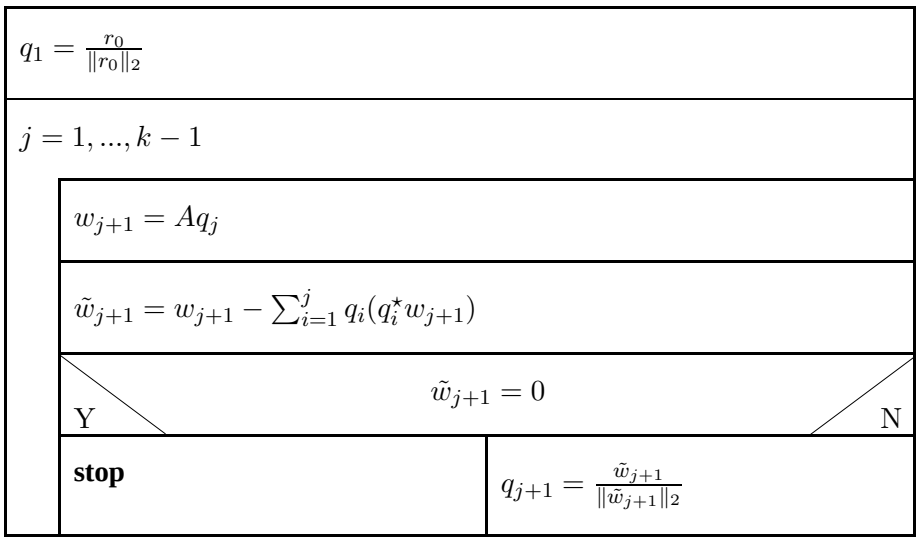

Abbildung 3.3: Arnoldi-Verfahren

Lanczos-Verfahren

Für den Spezialfall, dass $A = A^\star$ ist (A hermitesch), entspricht der Lanczos-Algorithmus im wesentlichen dem Arnoldi-Algorithmus. In diesem Fall ist die Hessenbergmatrix H tridiagonal und es kann eine 3-Term-Rekursion angewendet werden, das heißt es müssen nicht alle alten q_i gespeichert werden.

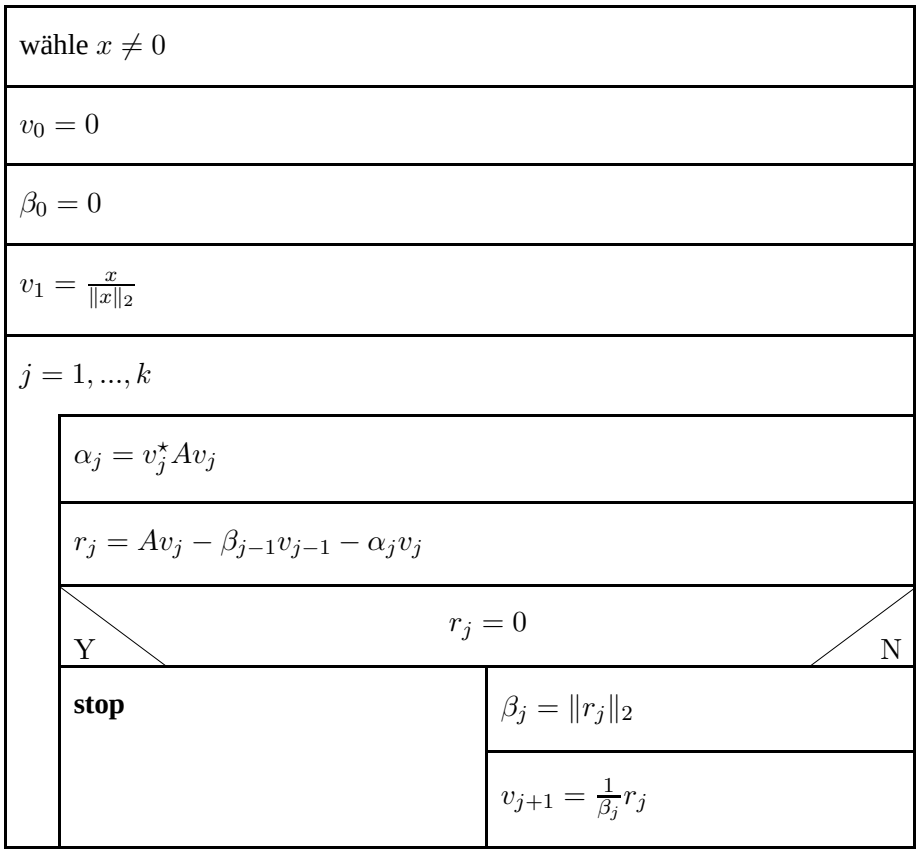

Abbildung 3.4: Symmetrisches Lanczos-Verfahren

Sowohl beim Arnoldi- als auch beim symmetrischen Lanczos-Verfahren kann ein vorzeitiger Stopp auftreten, was allerdings in der Praxis fast nie vorkommt. Tritt dieser Fall ein, ist der Krylov-Raum A-invariant und die Ritzpaare sind exakt.

3.3 CG-artige Verfahren

Symmetrische, positiv definite, lineare Gleichungssysteme können mit dem klassischen CG-Verfahren (**C**onjugate **G**radient) gelöst werden.

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig

$g^{(0)} = b - Ax^{(0)}, \;\; d^{(0)} = g^{(0)}$

$k = 0$

$\|g^{(k)}\| > \varepsilon$

- $\alpha_k = \frac{{g^{(k)}}^T g^{(k)}}{{d^{(k)}}^T A d^{(k)}}$
- $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$
- $g^{(k+1)} = g^k + \alpha_k A d^{(k)}$
- $\beta_k = \frac{{g^{(k+1)}}^T g^{(k+1)}}{{g^{(k)}}^T g^{(k)}}$
- $d^{k+1} = -g^{(k+1)} + \beta_k d^{(k)}$
- $k = k + 1$

Abbildung 3.5: CG-Verfahren

Sind die Matrizen indefinit oder unsymmetrisch, so konvergiert das klassische CG-Verfahren in der Regel nicht und man muss es modifizieren. CG-artige Verfahren, die auch dann konvergieren, sind das QMR- (**Q**uasi-**M**inimal **R**esidual) und TFQMR-Verfahren (**T**ranspose-**F**ree **Q**uasi-**M**inimal **R**esidual).

Das QMR-Verfahren kombiniert die Lanczos-Methode mit dem Ansatz der quasi-minimalen Residuen. Allerdings wird in jedem Schritt nicht die Norm der Residuen, sondern jeweils eine gewichtete Norm minimiert, wodurch sich der Aufwand des zu lösenden Minimierungsproblems stark verringert. Insgesamt ergibt sich daraus der folgende Algorithmus, der aus Gründen der Übersichtlichkeit in Pseudocode dargestellt ist:

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig

$g^{(0)} = Ax^{(0)} - b, \tilde{g}^{(1)} = -g^{(0)}$

löse $M_1 y = \tilde{g}^{(1)}$

$\rho_1 = \|y\|_2$

wähle $\tilde{w}^{(1)}$ beliebig, zum Beispiel: $\tilde{w}^{(1)} = -g^{(0)}$

löse $M_2^T t = \tilde{w}^{(1)}$

$\xi_1 = \|t\|_2, \gamma_0 = 1, \eta_0 = -1$

while $\|g^{(i)}\|_2 \leq \epsilon_r$

$q^{(i)} = \frac{\tilde{q}^{(i)}}{\rho_i}, y = \frac{y}{\rho_i}, w^{(i)} = \frac{\tilde{w}^{(i)}}{\xi_i}, t = \frac{t}{\xi_i}, \delta_i = t^T y$

löse $M_2 \tilde{y} = y$, löse $M_1^T \tilde{t} = t$

if $i = 1$ **then**

$p^{(1)} = \tilde{y}, v^{(1)} = \tilde{t}$

else

$p^{(1)} = \tilde{y} - \frac{\xi_i \delta_i}{\varepsilon_{i-1}} p^{(i-1)}, v^{(1)} = \tilde{t} - \frac{\rho_i \delta_i}{\varepsilon_{i-1}} v^{(i-1)}$

end if

$\varepsilon_i = {v^{(i)}}^T A p^{(i)}, \beta_i = \frac{\varepsilon_i}{\delta_i}, \tilde{q}^{(i+1)} = Ap^{(i)} - \beta_i q^{(i)}$

löse $M_1 y = \tilde{q}^{(i+1)}$

$\rho_{i+1} = \|y\|_2, \tilde{w}^{(i+1)} = A^T v^{(i)} - \beta_i w^{(i)}$

löse $M_2^T t = \tilde{w}^{(i+1)}$

$\xi_{i+1} = \|t\|_2, \vartheta_i = \frac{\rho_{i+1}}{\gamma_{i-1}|\beta_i|}, \gamma_i = \frac{1}{\sqrt{1+\vartheta_i^2}}, \eta_i = \frac{-\eta_{i-1}\rho_i\gamma_i^2}{\beta_i \gamma_{i-1}^2}$

if $i = 1$ **then**

$d^{(i)} = \eta_i p^{(i)}, s^{(i)} = \eta_i A p^{(i)}$

else

$d^{(i)} = \eta_i p^{(i)} + (\vartheta_{i-1}\gamma_i)^2 d^{(i-1)}, s^{(i)} = \eta_i A p^{(i)} + (\vartheta_{i-1}\gamma_i)^2 s^{(i-1)}$

end if

$x^{(i)} = x^{(i-1)} + d^{(i)}, g^{(i)} = g^{(i-1)} + s^{(i)}$

end while

Abbildung 3.6: QMR-Verfahren

Es handelt sich hierbei um ein vorkonditioniertes QMR-Verfahren mit Vorkonditionierer M. Für M_1 und M_2 gilt dabei: $M = M_1 M_2$.

Das QMR-Verfahren verwendet an einigen Stellen Matrix-Vektorprodukte mit transponierten Matrizen. Das Transponieren ist aber, insbesondere auf Parallelrechnern, eine sehr aufwändige Operation. Diese wird durch das TFQMR-Verfahren vermieden.

Der Algorithmus zur TFQMR-Methode sieht wie folgt aus:

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig

$g^{(0)} = Ax^{(0)} - b, \;\; y^{(1)} = -g^{(0)}, \;\; w^{(1)} = -g^{(0)}, \;\; v^{(0)} = Ay^{(1)}$

$d^{(0)} = 0, \;\; \tau_0 = \|g^{(0)}\|_2, \;\; \vartheta_0 = 0, \;\; \eta_0 = 0$

wähle $\tilde{g}^{(0)}$ so, dass $\rho_0 = -\tilde{g}^{(0)^T} g^{(0)} \neq 0$

$x^{(2i-1)}$ und $x^{(2i)}$ nicht konvergiert

- $\sigma_{i-1} = \tilde{g}^{(0)^T} v^{(i-1)}, \;\; \alpha_{i-1} = \frac{\rho_{i-1}}{\sigma_{i-1}}, \;\; y^{(2i)} = y^{(2i-1)} - \alpha_{i-1} v^{(i-1)}$
- $j = 2i-1, 2i$
  - $w^{(j+1)} = w^{(j)} - \alpha_{i-1} A y^{(j)}, \;\; \vartheta_j = \frac{\|w^{(j+1)}\|_2}{\tau_{j-1}}$
  - $\gamma_j = \frac{1}{\sqrt{1+\theta_j^2}}, \;\; \tau_j = \tau_{j-1} \vartheta_j \gamma_j, \;\; \eta_j = \gamma_j^2 \alpha_{i-1}$
  - $d^{(j)} = y^{(j)} + \frac{\vartheta_{j-1}^2 \eta_{j-1}}{\alpha_{i-1}} d^{(j-1)}, \;\; x^{(j)} = x^{(j-1)} + \eta_j d^{(j)}$
- $\rho_i = \tilde{g}^{(0)^T} w^{(2i+1)}, \;\; \beta_i = \frac{\rho_i}{\rho_{i-1}}, \;\; y^{(2i+1)} = w^{(2i+1)} \beta_i y^{(2i)}$
- $v^{(i)} = Ay^{(2i+1)} + \beta_i (Ay^{(2i)} + \beta_i v^{(i-1)})$

Abbildung 3.7: TFQMR-Verfahren

Kapitel 4

Das Jacobi-Davidson-Verfahren

Das Jacobi-Davidson-Verfahren ist eine Kombination aus den Hauptideen des Jacobi- und des Davidson-Verfahrens, die am Anfang dieses Kapitels vorgestellt werden (siehe hierzu [15]).

4.1 Das Davidson-Verfahren

Beim Davidson-Verfahren geht man von einem Unterraum U der Dimension k mit einer orthonormalen Basis $u_1, ..., u_k$ aus. Sei U_k die Matrix mit den Spalten $u_1, ..., u_k$, dann lässt sich die Matrix A reduziert auf den Unterraum schreiben als $B_k = U_k^\star A U_k$, das heißt B_k ist eine Galerkin-Projektion von A.
Sei (θ, s) ein Eigenpaar von B_k, dann gilt:

$$B_k s = \theta s$$

Die Eigenwerte der Matrix B_k sind die Ritzwerte der Matrix A bezüglich U_k und $y = U_k s$ liefert den Ritzvektor von A zu einem Eigenvektor s der Matrix B_k, denn θ und $U_k s$ erfüllen die Ritz-Galerkin-Bedingung für das Residuum des angenäherten Eigenpaares:

$$\begin{aligned} & r = AU_k s - \theta U_k s \quad \perp \quad \{u_1, ..., u_k\} \\ \Leftrightarrow \quad & U_k^\star r = U_k^\star A U_k s - \theta s \\ & = B_k s - \theta s \\ & = 0 \end{aligned} \tag{4.1}$$

Daraus lässt sich ableiten:

$$\begin{aligned} r &= AU_k s - \theta U_k s \\ &= Ay - \theta y \\ \Leftrightarrow \quad y &= (A - \theta I)^{-1} r \end{aligned}$$

Davidson arbeitete als Chemiker hauptsächlich mit streng diagonal dominanten Matrizen (siehe auch [3]). Bei diesen kann man den "Shift-and-Invert"-Schritt $(A - \theta I)^{-1}$ annähern durch $(D_A - \theta I)^{-1}$, wobei D_A die Diagonale von A darstellt.
Dadurch erhält man einen neuen Vektor

$$t = (D_A - \theta I)^{-1} r$$

der zur Erweiterung des Unterraums verwendet wird. Der Vektor t wird bezüglich des Unterraums U_k orthonormalisiert und zur bestehenden Basis hinzugefügt.

Verwendet man hierbei keine Näherung für $A - \theta I$, so erhält man für t beziehungsweise y den letzten Basisvektor u_k. Damit kann der Unterraum nicht erweitert werden.

Betrachtet man extrem diagonal dominante Matrizen, das heißt Matrizen, deren Diagonalelemente wesentlich größer sind als die Nebendiagonalelemente, so stagniert das Verfahren. Dies lässt sich leicht erkennen, indem man von einer Diagonalmatrix A ausgeht.
Es ergibt sich:

$$\begin{aligned} r &= (A - \theta I)y \\ &= (D_A - \theta I)y \end{aligned}$$

Die Matrix $D_A - \theta I$ enhält hierbei mindestens eine Zeile, die vollständig Null ist. Daher ist die Matrix nicht invertierbar und somit kann der Unterraum nicht mehr erweitert werden.

Ursprünglich wurde das Davidson-Verfahren für symmetrische Matrizen entwickelt. Es hat sich aber gezeigt, dass es ebenso für unsymmetrische Matrizen geeignet ist. In Abbildung 4.1 ist das Davidson-Verfahren für unsymmetrische Matrizen im Detail dargestellt. Bei symmetrischen Matrizen müssen nur ungefähr die Hälfte an inneren Produkten bei der Konstruktion der B_k berechnet werden.

$u_1 = \frac{v}{\|v\|_2}$

$k = 1, ..., m$

- $w_k = Au_k$
- berechne die k-te Zeile und Spalte von B_k
- $j = 1, ..., k-1$
  - $B_{j,k} = u_j^\star w_k$
  - $B_{k,j} = u_k^\star w_j$
- $B_{k,k} = u_k^\star w_k$
- berechne den größten Eigenwert θ von B_k und den zugehörigen Eigenvektor s mit Komponenten $s_{(j)}$
- $z = \sum_{j=1}^{k} s_{(j)} u_j$
- $r = Az - \theta z$
- $t = (D_A - \theta I)^{-1} r$
- $j = 1, ..., k$
  - $t = t - (u_j^\star t) u_j$
- $u_{k+1} = \frac{t}{\|t\|_2}$

Abbildung 4.1: Davidson-Verfahren mit Diagonal-Vorkonditionierer

4.2 Das Verfahren von Jacobi

Das Verfahren von Jacobi (siehe hierzu [6]) gliedert sich in zwei Schritte. Im ersten Schritt verwendet man das klassische Jacobi-Verfahren zur Eigenwertberechnung symmetrischer Matrizen, um die Matrix A auf die Form $\begin{pmatrix} \alpha & c^T \\ b & F \end{pmatrix}$ zu bringen, indem man Rotationen durchführt. Als zweiter Schritt wird ein lineares System für das orthogonale Komplement der genäherten Eigenvektoren abgeleitet, welches dann mit Gauss-Jacobi-Iterationen gelöst wird. Die Konstruktion des linearen Systems wird auch als *"Jacobi's orthogonal complement correction" (JOCC)* bezeichnet (siehe hierzu auch [13]).

Seien A eine streng diagonal dominante Matrix und ohne Beschränkung der Allgemeinheit $a_{1,1} = \alpha$ das größte Diagonalelement, dann ist α auch eine Näherung für den größten Eigenwert λ und der erste Einheitsvektor e_1 eine Näherung für den zugehörigen Eigenvektor u. Damit ergibt sich als Eigenwertproblem:

$$A \begin{pmatrix} 1 \\ z \end{pmatrix} = \begin{pmatrix} \alpha & c^T \\ b & F \end{pmatrix} \begin{pmatrix} 1 \\ z \end{pmatrix} = \lambda \begin{pmatrix} 1 \\ z \end{pmatrix} \tag{4.2}$$

Dabei sind F eine quadratische Matrix, α ein Skalar und b, c und z Vektoren der entsprechenden Größe. Gesucht sind nun der Eigenwert λ, der nahe bei α liegt, und der zugehörige Eigenvektor

$$u = \begin{pmatrix} 1 \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ z \end{pmatrix} + e_1$$

Das Eigenwertproblem lässt sich anders schreiben als:

$$\begin{aligned} \lambda &= \alpha + c^T z \\ (F - \lambda I)z &= -b \end{aligned} \tag{4.3}$$

Gleichung (4.3) lässt sich mit D als Diagonale von F anders darstellen als:

$$\begin{aligned} & (F - \lambda I)z &&= -b \\ \Leftrightarrow \quad & (D - D + F - \lambda I)z &&= -b \\ \Leftrightarrow \quad & (D - \lambda I)z &&= (D - F)z - b \end{aligned}$$

Daraus läßt sich folgende Gauss-Jacobi-Iterationsvorschrift mit $z_1 = 0$ herleiten:

$$\begin{cases} \theta_k &= \alpha + c^T z_k \\ (D - \theta_k I) z_{k+1} &= (D - F) z_k - b \end{cases}$$

wobei θ_k eine Näherung für λ und z_k eine Näherung für z ist.

4.3 Das Jacobi-Davidson-Verfahren

Das Jacobi- und das Davidson-Verfahren versuchen, eine gegebene Eigenvektornäherung zu korrigieren. Man sucht das orthogonale Komplement zur aktuellen Näherung u_k des gesuchten Eigenvektors u von A.

Durch Jacobis Idee, die Komponente des gesuchten Eigenvektors senkrecht auf den Startvektor e_1 zu berechnen, betrachtet man das gegebene System ohne die erste Zeile und Spalte. Dies entspricht einer Projektion des gegebenen Eigenwertproblems auf das orthogonale Komplement von e_1. Da man an der Komponente von x senkrecht auf die momentane Näherung u_k für x interessiert ist, beschränkt man sich auf den Unterraum $u_k^{\perp}$.

Die orthogonale Projektion von A auf den Unterraum $u_k^{\perp}$ ist gegeben durch:

$$B = (I - u_k u_k^{\star}) A (I - u_k u_k^{\star}) \tag{4.4}$$

Dabei nimmt man an, dass u_k normalisiert ist. Für den Ritzwert θ_k gilt:

$$\begin{aligned} & \theta_k u_k = A u_k \\ \Leftrightarrow \quad & \theta_k u_k^{\star} u_k = u_k^{\star} A u_k \\ \Leftrightarrow \quad & \theta_k = u_k^{\star} A u_k \end{aligned} \tag{4.5}$$

Daher lässt sich die Projektionsvorschrift (4.4) wie folgt umformen:

$$\begin{aligned} & B = (I - u_k u_k^{\star}) A (I - u_k u_k^{\star}) \\ \Leftrightarrow \quad & B = (A - u_k u_k^{\star} A)(I - u_k u_k^{\star}) \\ \Leftrightarrow \quad & B = A - u_k u_k^{\star} A - A u_k u_k^{\star} + u_k (u_k^{\star} A u_k) u_k^{\star} \\ \Leftrightarrow \quad & A = B + u_k u_k^{\star} A + A u_k u_k^{\star} - \theta_k u_k u_k^{\star} \end{aligned} \tag{4.6}$$

Möchte man nun das Eigenpaar (λ, x) von A berechnen, so weiß man, dass θ_k nahe bei λ und u_k nahe bei x liegen. Weiterhin erfüllen ν mit $\nu \perp u_k$ und $x = u_k + \nu$ die Bedingung:

$$A(u_k + \nu) = \lambda(u_k + \nu) \tag{4.7}$$

Des Weiteren gilt für die Matrix B:

$$\begin{aligned} B u_k &= (I - u_k u_k^{\star}) A (I - u_k u_k^{\star}) u_k \\ &= (I - u_k u_k^{\star}) A (u_k - u_k) \\ &= 0 \end{aligned} \tag{4.8}$$

Aus (4.7) folgt unter Berücksichtigung von (4.5),(4.6) und (4.8):

$$\begin{aligned} & (B + u_k u_k^{\star} A + A u_k u_k^{\star} - \theta_k u_k u_k^{\star})(u_k + \nu) = \lambda(u_k + \nu) \\ \Leftrightarrow \quad & 0 + B\nu + u_k \theta_k + u_k u_k^{\star} A \nu + A u_k + 0 - \theta_k u_k + 0 = \lambda u_k + \lambda \nu \\ \Leftrightarrow \quad & B\nu - \lambda\nu = -(A u_k - \theta_k u_k) + (\lambda - \theta_k - u_k^{\star} A \nu) u_k \\ \Leftrightarrow \quad & (B - \lambda I)\nu = -r_k + (\lambda - \theta_k - u_k^{\star} A \nu) u_k \end{aligned} \tag{4.9}$$

Da $(B - \lambda I)\nu$ und r_k beide senkrecht zu u_k sind, muss der Vorfaktor von u_k ebenfalls gleich Null sein. Somit gilt:

$$(B - \lambda I)\nu = -r_k = (A - \theta_k I) u_k \tag{4.10}$$

Da der Wert von λ nicht bekannt ist, wird er durch die momentane Näherung θ_k ersetzt. Falls θ_k nicht nahe genug am gesuchten Eigenwert liegt, ist es häufig effizienter, λ durch einen festen Wert

zu ersetzen, der nahe beim gesuchten Eigenwert liegt. Dadurch wird eine Konvergenz gegen einen nicht gewünschten Eigenwert oder eine lokale Stagnation verhindert. Dies führt mit Hilfe von (4.4) und der Tatsache

$$(I - u_k u_k^\star)\nu = \nu$$

zu der Korrekturgleichung:

$$\begin{aligned}
& [(I - u_k u_k^\star)A(I - u_k u_k^\star) - \theta_k I]\nu &&= -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^\star)A(I - u_k u_k^\star)\nu - \theta_k I\nu &&= -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^\star)A\nu - \theta_k(I - u_k u_k^\star)\nu &&= -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^\star)(A - \theta_k I)\nu &&= -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^\star)(A - \theta_k I)(I - u_k u_k^\star)\nu &&= -r_k
\end{aligned} \tag{4.11}$$

Diese Korrekturgleichung bildet die Grundlage für das von *Sleijpen* und *Van der Vorst* entwickelte Jacobi-Davidson-Verfahren (siehe [13]). Da es sich bei θ_k bereits um eine Näherung für den Eigenwert λ handelt, kann man statt mit $A - \theta_k I$ auch mit einer Näherung K für $A - \theta_k I$ arbeiten.

Für einen gegebenen Ritzwert θ_k wird aus (4.11) ein ν berechnet, mit dem der Unterraum erweitert wird. Dann berechnet man ein neues Ritzpaar in Bezug auf den erweiterten Unterraum. Diese Schritte werden wiederholt, bis die Eigenwertnäherung exakt genug ist.

Arbeitet man mit Vorkonditionierern, so ist es vorteilhaft Gleichung (4.11) weiter umzuformen:

$$\begin{aligned}
& (I - u_k u_k^\star)(A - \theta_k I)(I - u_k u_k^\star)\nu &&= -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^\star)(A - \theta_k I)\nu &&= -r_k \\
\Leftrightarrow \quad & (A - \theta_k I)\nu &&= -r_k + \alpha u_k \quad \text{mit} \quad \alpha = u_k^\star(A - \theta_k I)\nu
\end{aligned}$$

Da ν unbekannt ist, kann man zunächst auch α nicht berechnen. Durch einige Umformungen wird dies aber möglich. Sei dafür K eine Näherung für die Matrix $A - \theta_k I$, dann lässt sich ν aus folgender Formel berechnen:

$$K\nu = -r_k + \alpha u_k$$

Durch einige Umformungen erhält man unter Beachtung, dass $u_k \perp \nu$ ist, eine Berechnungsvorschrift für α:

$$\begin{aligned}
\nu &= -K^{-1}r_k + \alpha K^{-1}u_k \\
\Leftrightarrow \quad 0 &= -u_k^\star K^{-1}r_k + \alpha u_k^\star K^{-1}u_k \\
\Leftrightarrow \quad \alpha &= \frac{u_k^\star K^{-1}r_k}{u_k^\star K^{-1}u_k}
\end{aligned}$$

Da zwei Operationen mit K^{-1} durchgeführt werden müssen, ist der Aufwand sehr hoch. Die Berechnungen können jedoch auf eine Operation mit K^{-1} reduziert werden, wenn man (4.11) mit einem iterativen Löser vorkonditioniert.

Bei einem direkten Löser, bei dem eine Zerlegung durchgeführt wird, macht es keinen großen Unterschied, ob die bestimmte Zerlegung auf eine oder zwei rechte Seiten angewendet wird.

Geht man von $K = A - \theta_k I$ aus, so erhält man unter Verwendung von (4.10) formal:

$$\begin{aligned}
(A - \theta_k I)\nu &= -r_k + \alpha u_k \\
\Leftrightarrow \quad \nu &= -(A - \theta_k I)^{-1}r_k + \alpha(A - \theta_k I)^{-1}u_k \\
&= -u_k + \alpha(A - \theta_k I)^{-1}u_k
\end{aligned}$$

Da u_k bereits im Unterraum liegt, wird dieser effektiv durch den normalisierten Vektor $(A - \theta_k I)^{-1}u_k$ erweitert.

Dies ist derselbe Vektor, der von der Rayleigh-Quotienten-Iteration erzeugt wird. Bei exakter Arithmetik erhält man also eine beschleunigte Rayleigh-Quotienten-Iteration (RQI) für die invertierte charakteristische Matrix. Von dieser Methode ist bekannt, dass sie für symmetrische Matrizen kubische und für unsymmetrische quadratische Konvergenz aufweist (siehe hierzu [12]). Die quadratische Konvergenz des Jacobi-Davidson-Verfahrens lässt sich auch erkennen, wenn man es als abgeleitete Newton-Iteration betrachtet (siehe hierzu [15]). Das Jacobi-Davidson-Verfahren kann noch beschleunigt werden, wenn die Korrekturgleichung (4.11) nur näherungsweise gelöst wird. Es geht dabei allerdings in der Regel die kubische beziehungsweise quadratische Konvergenz verloren.

Zusammenfassung

Zusammenfassend kann man das Jacobi-Davidson-Verfahren als Kombination zweier Grundideen beschreiben:

- Zuerst wird mit Hilfe der Ritz-Galerkin-Bedingung (siehe Gleichung (4.1) auf Seite 18) eine Näherung für das Eigenwertproblem $Ax = \lambda x$ im aktuellen Unterraum bestimmt. Als Lösung erhält man k Paare $(\theta_j^{(k)}, u_j^{(k)} := U_k s_j^{(k)})$, die Ritzwerte und Ritzvektoren von A bezüglich des Unterraums.
- Mit Hilfe der Korrekturgleichung (4.11) wird dann das orthogonale Komplement zu dem Ritzvektor, dessen Ritzwert am nächsten am gewünschten Ziel liegt, bestimmt, mit dem dann der Unterraum erweitert wird.

Im Verfahren wird Jacobis Ansatz für die Suche nach dem orthogonalen Komplement verwendet. Weiterhin bedient man sich des Davidson-Verfahrens, um die Erweiterung des Unterraums durchzuführen.

Zu beachten bleibt aber, dass man die Korrekturgleichung (4.11) nicht notwendigerweise mit dem von Davidson verwendeten “Shift-and-Invert“-Verfahren lösen muss. Dies bedeutet weiterhin, dass man nicht mehr länger auf diagonal dominante Matrizen beschränkt ist.

Löst man die Korrekturgleichung (4.11) näherungsweise, so läuft dies formal für spezielle Näherungen auf bereits bekannte Verfahren hinaus (siehe auch [17]):

- Nimmt man die sehr grobe Näherung $\nu = -r_k$, so ist das Verfahren äquivalent zur Arnoldi-Methode beziehungsweise für symmetrisches A zum Lanczos-Verfahren.
- Nähert man den Operator $A - \theta_k I$ durch $D_A - \theta_k I$ an und vernachlässigt die Orthogonalitätsbedingung $\nu \perp u_k$, so erhält man das Davidson-Verfahren.

In der Abbildung 4.2 wird das Jacobi-Davidson-Verfahren im Detail beschrieben. Es handelt sich um eine Form des Algorithmus, die nur den größten Eigenwert λ_{max} von A sucht.

setze t auf Startnäherung $t = v_0$

$m = 1, ...$

- $i = 1, ..., m-1$
  - $t = t - < v_i, t > v_i$
- $v_m = \frac{t}{\|t\|_2}$
- $v_m^A = Av_m$
- $i = 1, ..., m-1$
  - $M_{i,m} = v_i^\star v_m^A$
  - $M_{m,i} = v_m^\star v_i^A$
- $M_{m,m} = v_m^\star v_m^A$
- berechne das größte Eigenpaar $Ms = \theta s$ von M mit $\|s\|_2 = 1$
- $u = Vs$ mit $V = (v_1, ..., v_n)$
- $u^A = V^A s$ mit $V^A = (v_1^A, ..., v_n^A)$
- $r = u^A - \theta u$
- $\|r\|_2 \leq \epsilon$
  - Y:
    - $\tilde{\lambda} = \theta$
    - $\tilde{x} = u$
    - **stop**
  - N:
- berechne näherungsweise $t \perp u$ aus $(I - uu^\star)(A - \theta I)(I - uu^\star)t = -r$

Abbildung 4.2: Einfacher Jacobi-Davidson-Algorithmus für λ_{max} von A

4.4 Iterative Lösung der Korrekturgleichung

Sei K ein Vorkonditionierer für die Matrix $A - \theta_j^{(k)} I$, so dass gilt:

$$K^{-1}(A - \theta_j^{(k)} I) \approx I$$

Der Operator K muss auf den Unterraum senkrecht zu $u_j^{(k)}$ beschränkt sein, weshalb man mit folgendem Vorkonditionierer arbeitet:

$$\tilde{K} := (I - u_j^{(k)} u_j^{(k)^\star}) K (I - u_j^{(k)} u_j^{(k)^\star})$$

Dies bedeutet viel Mehraufwand, da die durchzuführenden Projektionen $I - u_j^{(k)} u_j^{(k)^\star}$ sehr aufwändig sind. Dies kann jedoch auf einige einfache Operationen zurückgeführt werden.

Es wird im Folgenden davon ausgegangen, dass ein Krylov-Löser mit linksseitigem Vorkonditionierer und Startwert $t_0 = 0$ für die Lösung der Korrekturgleichung (4.11) verwendet wird. Da der Startvektor im Unterraum liegt und $\tilde{K}^{-1}$ wieder dorthin projiziert, werden auch alle Iterationsvektoren des Krylov-Lösers im Unterraum liegen.
Im jeweiligen Unterraum berechnet man nun den Vektor $z := \tilde{K}^{-1}\tilde{A}\nu$ für einen Vektor ν aus dem Krylov-Unterraum und die Matrix

$$\tilde{A} = (I - u_j^{(k)} u_j^{(k)^\star})(A - \theta_j^{(k)} I)(I - u_j^{(k)} u_j^{(k)^\star})$$

Dies geschieht in zwei Schritten.
Zunächst berechnet man:

$$\begin{aligned} \tilde{A}\nu &= (I - u_j^{(k)} u_j^{(k)^\star})(A - \theta_j^{(k)} I)(I - u_j^{(k)} u_j^{(k)^\star})\nu \\ &= (I - u_j^{(k)} u_j^{(k)^\star}) y \quad \text{mit } y = (A - \theta_j^{(k)} I)\nu \end{aligned}$$

Nun lässt sich $z \perp u_j^{(k)}$ berechnen durch:

$$\begin{aligned} & & \tilde{K} z &= (I - u_j^{(k)} u_j^{(k)^\star}) y \\ \Leftrightarrow & (I - u_j^{(k)} u_j^{(k)^\star}) K (I - u_j^{(k)} u_j^{(k)^\star}) z &= (I - u_j^{(k)} u_j^{(k)^\star}) y \\ \Rightarrow & K (I - u_j^{(k)} u_j^{(k)^\star}) z &= y \\ \Leftrightarrow & K z &= y - \alpha u_j^{(k)} \end{aligned}$$

Formt man diese Gleichung weiter um, so führt dies zu einer Berechnungsvorschrift für α:

$$\begin{aligned} z &= K^{-1} y - \alpha K^{-1} u_j^{(k)} \\ \Leftrightarrow \quad 0 &= u_j^{(k)^\star} K^{-1} y - \alpha u_j^{(k)^\star} K^{-1} u_j^{(k)} \\ \Leftrightarrow \quad \alpha &= \frac{u_j^{(k)^\star} K^{-1} y}{u_j^{(k)^\star} K^{-1} u_j^{(k)}} \end{aligned}$$

Bei i_s Iterationen des linearen Lösers werden nur $i_s + 1$ Operationen mit K^{-1} durchgeführt, da nur einmal pro Iteration $K^{-1} u_j^{(k)}$ berechnet werden muss. Weiterhin benötigt man für eine Matrix-Vektormultiplikation $\tilde{K}^{-1}\tilde{A}$ nur ein inneres Produkt und ein Vektor-Update anstatt vier Berechnungen mit dem Projektor $I - u_j^{(k)} u_j^{(k)^\star}$.

In der folgenden Abbildung ist das Verfahren zur Lösung der Korrekturgleichung (4.11) mit einer Krylov-Unterraummethode und Vorkonditionierung im Detail beschrieben.

$\hat{u} = K^{-1}u$
$\mu = u^{\star}\hat{u}$
$\hat{r} = K^{-1}r$
$\tilde{r} = \hat{r} - \frac{u^{\star}\hat{r}}{\mu}\hat{u}$
berechne ν durch Krylov-Löser $\tilde{K}^{-1}\tilde{A}\nu = -\tilde{r}$
$y = (A - \theta I)\nu$
$\hat{y} = K^{-1}y$
$z = \hat{y} - \frac{u^{\star}\hat{y}}{\mu}\hat{u}$

Abbildung 4.3: Lösung der Korrekturgleichung mit Vorkonditionierung

4.5 Restart-Strategien

Für ein Eigenpaar

Der Mehraufwand, der bei der Vergrößerung des Unterraums entsteht, macht einen sogenannten Restart notwendig, zum Beispiel, wenn der Speicher nicht mehr ausreicht. Außerdem wird das Lösen des projizierten Eigenwertproblems aufwändiger, je größer der Unterraum wird, und somit wird das Verfahren ineffektiver. Ein offensichtlicher Weg für einen Restart ist, die letzten Näherungen für den gewünschten Eigenvektor zu nehmen, auch wenn dies nicht notwendigerweise die effizienteste Methode ist. Bei jedem Restart mit nur einem Vektor lässt man mögliche wertvolle Informationen über den Unterraum fallen, die im verbleibenden Teil des Unterraumes enthalten sind. Trotzdem haben wir einen brauchbaren Unterraum, in dem alle Vektoren Informationen über den gewünschten Eigenvektor enthalten.

Der Informationsverlust bei einem Restart äußert sich in einer Verringerung der Konvergenzgeschwindigkeit. Daher ist es oft besser, mit einer Menge von Approximationen für Eigenvektoren den Restart durchzuführen, die einen Unterraum aufspannen, der mehr Informationen für das gewünschte Eigenpaar enthält anstatt den Restart mit nur einem Vektor zu beginnen. Eine gute Strategie ist hier, den Restart mit dem Unterraum durchzuführen, der durch eine kleine Menge von Ritzvektoren aufgespannt wird, die am nächsten am gegebenen Zielvektor liegen, das heißt deren Residuen klein sind.

Für mehrere Eigenpaare

Unter gewissen Umständen hat das Jacobi-Davidson-Verfahren einige sichtbare Nachteile gegenüber dem Standard-Arnoldi-Verfahren. Das Jacobi-Davidson-Verfahren konvergiert sehr schnell gegen einen Eigenwert. Für das Arnoldi-Verfahren ist es kein großes Problem, auch mehrere Eigenwerte zu berechnen, da die gewöhnlich langsamere Konvergenz von Ritzwerten zu einem speziellen Eigenwert Hand in Hand geht mit der gleichzeitigen Konvergenz von anderen Ritzwerten zu anderen Eigenwerten. Jacobi-Davidson zielt absichtlich nur auf einen speziellen Eigenwert ab, der am nächsten bei einem vorgegebenen Ziel liegt. Das Arnoldi-Verfahren hingegen führt zu Näherungen für eventuell alle Eigenwerte. In der frühen Phase der Iteration führt es gewöhnlich zu isolierten äußeren Eigenwerten.

Für das Jacobi-Davidson-Verfahren würde man einen Restart mit unterschiedlichen ausgewählten Ritzpaaren durchführen. Es ist nicht garantiert, dass dies zu einem neuen Eigenpaar führt. Ein weiteres Problem stellen mehrfache Eigenwerte dar, aber dieses Problem besteht auch bei anderen Unterraumverfahren.

Ein bekannter Weg, um dieses Problem zu beheben, ist die ***Deflation*-Strategie**. Konvergiert das Verfahren gegen einen Eigenvektor, so wird die Iteration in einem Unterraum fortgeführt, der von den restlichen Eigenvektoren aufgespannt wird. Für hermitesche Matrizen A stellt dies kein Problem dar, da sie ein orthonormales Eigenvektorsystem besitzen. Da verhindert werden soll, dass mit nicht orthogonalen Reduktions- beziehungsweise Deflationsoperatoren gearbeitet wird, sollte man für nicht-hermitesche Matrizen mit sogenannten ***Schurvektoren*** arbeiten. Diese Schurvektoren sind definiert als die Spalten einer orthogonalen Matrix Q, die A in eine obere Dreiecksform transformiert:

$$Q^\star AQ = R$$

Dadurch erhält man die Eigenwerte von A auf der Diagonalen von R und die Eigenvektoren x von A können aus den Eigenvektoren y von R durch $x = Qy$ berechnet werden.

Es existiert ein Algorithmus zur Berechnung mehrerer Eigenpaare. Dieser basiert auf der Berechnung einer Schurform von A:

$$AQ_k = Q_k R_k$$

Dabei ist Q_k eine orthonormale $n \times k$ Matrix, R_k eine obere $k \times k$ Dreiecksmatrix, wobei $k << n$ gilt. Ist nun (λ, x) ein Eigenpaar von R_k, so ist $(\lambda, Q_k x)$ ein Eigenpaar von A.

Zur Berechnung der Schurform für Eigenwerte nahe einem bestimmten Wert τ sind folgende Schritte erforderlich:

1. Gegeben sei ein orthonormales Unterraumbasissystem $v_1, ..., v_i$ mit zugehöriger Matrix V_i, dann berechnet man die Projektionsmatrix $M = V_i^\star A V_i$. Dabei ist M eine $i \times i$ Matrix. Für M berechnet man anschließend die vollständige Schurform $MU = US$, wobei $U^\star U = I$ gilt und S eine obere Dreiecksmatrix ist. Dies kann man zum Beispiel über eine QR-Zerlegung machen.

   Dann ordnet man S so um, dass $|S_{i,i} - \tau|$ eine aufsteigende Reihe für wachsendes i bildet. Die ersten Diagonalelemente von S repräsentieren dabei Eigenwertnäherungen nahe an τ und die ersten entsprechenden Spalten von V_i stellen den Unterraum der besten Eigenvektornäherungen dar. Die Umordnung kann unter Beibehaltung der Dreiecksgestalt von S erfolgen

(siehe hierzu [1]).

Steht nur begrenzter Speicherplatz zur Verfügung, so kann man diesen Unterraum auch für den Restart nutzen, während man die restlichen Spalten einfach ignoriert. Der restliche Unterraum wird wie beim Jacobi-Davidson-Verfahren erweitert. Nachdem diese Prozedur konvergiert ist, erhält man ein Eigenpaar (λ, q) von A.

2. Angenommen, es ist bereits eine Schurform der Dimension k vorhanden und diese soll mit einer neuen Spalte q erweitert werden:

$$A[Q_k, q] = [Q_k, q] \begin{bmatrix} R_k & s \\ & \lambda \end{bmatrix}$$

Dabei ist $Q_k^\star q = 0$. Nach einigen Berechnungen erhält man hier:

$$(I - Q_k Q_k^\star)(A - \lambda I)(I - Q_k Q_k^\star)q = 0$$

Dies sagt aus, dass ein neues Paar (λ, q) ein Eigenpaar von

$$\tilde{A} = (I - Q_k Q_k^\star)A(I - Q_k Q_k^\star)$$

ist. Dieses Paar kann man durch Anwendung des Jacobi-Davidson-Verfahrens (mit Schurform-Reduktion wie in Schritt 1 für $\tilde{A}$) lösen.

Man sieht, dass nach jeder Konvergenz gegen ein Eigenpaar die Matrix $\tilde{A}$ zu aufwändigeren Berechnungen führt, aber es lässt sich zeigen, dass das gesamte Verfahren trotzdem effizient ist. Eine Erklärung hierfür ist, dass nach der Konvergenz einiger Eigenvektoren, die zu Eigenwerten nahe dem Ziel τ gehören, die Matrix $\tilde{A}$ besser konditioniert ist. Das bedeutet, dass die Korrekturgleichung im Jacobi-Davidson-Verfahren einfacher von einem iterativen Löser gelöst werden kann.

Es ist möglich, diese Korrekturgleichung mit einem vorkonditionierten iterativen Löser zu lösen. Da derselbe Vorkonditionierer für verschiedene Eigenpaare verwendet werden kann, lohnt es sich, gute Vorkonditionierer zu entwickeln.

Mit der zuvor erwähnten Restart-Strategie kann man erwarten, dass, wenn ein Ritzwert nahe genug an einem Eigenwert liegt, der verbleibende Teil des Unterraums schon viele Komponenten in der Nähe der Eigenpaare besitzt. Dies ist der Fall, da man in jedem Schritt die Ritzvektoren, die nahe an dem gewünschten Eigenwert lagen, genommen hat. Diese Information kann man als Basis für einen Unterraum zur Berechnung des nächsten Eigenvektors nutzen, nachdem die Eigenwertgleichung geeignet umgeformt wurde, um zu verhindern, dass der gerade gefundene Eigenvektor wieder in den Berechnungsprozeß miteingeht.

Seien $q_1, ..., q_{k-1}$ die akzeptierten Schurvektoren und seien diese Vektoren in guter Genauigkeit orthonormal, dann kann das Verfahren mit genäherten Schurvektoren mit Residuum $\|Aq_j - QRe_j\|_2$ innerhalb einer Toleranzgrenze ϵ einen Fehler der Ordnung von ϵ^2 in den Eigenwerten hervorrufen. Voraussetzung ist dabei, dass die berechneten Eigenwerte getrennt von den restlichen Eigenwerten liegen.
Die Matrix Q_{k-1} hat die Spaltenvektoren q_j. Daher wendet man den Jacobi-Davidson-Algorithmus auf die Matrix

$$(I - Q_{k-1} Q_{k-1}^\star)A(I - Q_{k-1} Q_{k-1}^\star)$$

an, um den nächsten Schurvektor q_k zu erhalten. Dies führt zu einer Korrekturgleichung der Form:

$$P_m(I - Q_{k-1} Q_{k-1}^\star)(A - \theta_j^{(m)} I)(I - Q_{k-1} Q_{k-1}^\star)P_m t_j^{(m)} = -r_j^{(m)}$$

Dabei ist $P_m := I - u_j^{(m)} u_j^{(m)\star}$. Dies muss für die Korrektur $t_j^{(m)}$ für jede neue Schurnäherung $u_j^{(m)}$ mit zugehörigem Ritzwert $\theta_j^{(m)}$ gelöst werden.

4.6 Vorkonditionierung

Sei nun K ein Vorkonditionierer von links für den Operator $A - \theta_j^{(m)} I$ und $\tilde{Q}$ die Matrix Q_{k-1} erweitert um die k-te Spalte $u_k^{(m)}$. In diesem Fall muss der Vorkonditionierer K auf den Unterraum orthogonal zu $\tilde{Q}$ beschränkt sein. Dies bedeutet, dass man effektiv mit

$$\tilde{K} \;:=\; (I - \tilde{Q}\tilde{Q}^\star)K(I - \tilde{Q}\tilde{Q}^\star)$$

arbeitet.

Im Folgenden wird nun davon ausgegangen, dass ein Krylov-Löser mit Startwert $t_0 = 0$ und ein Linksvorkonditionierer für die Näherungslösung der Korrekturgleichung verwendet wird. Da der Startvektor im Unterraum orthogonal zu $\tilde{Q}$ liegt und $\tilde{K}^{-1}$ dorthin projiziert, sind alle Iterationsvektoren des Krylov-Lösers auch in diesem Unterraum.
Hier muss der Vektor $z := \tilde{K}^{-1}\tilde{A}\nu$ berechnet werden. Dabei wird der Vektor ν durch den Krylov-Löser bereitgestellt. Weiterhin gilt:

$$\tilde{A} \;:=\; (I - \tilde{Q}\tilde{Q}^\star)(A - \theta_j^{(m)} I)(I - \tilde{Q}\tilde{Q}^\star)$$

Die Lösung erfolgt in zwei Schritten.
Erst berechnet man:

$$\begin{aligned} \tilde{A}v &= (I - \tilde{Q}\tilde{Q}^\star)(A - \theta_j^{(m)} I)(I - \tilde{Q}\tilde{Q}^\star)\nu \\ &= (I - \tilde{Q}\tilde{Q}^\star)y \text{ mit } y := (A - \theta_j^{(m)} I)\nu \end{aligned}$$

Danach löst man durch Vorkonditionierung $z \perp \tilde{Q}$ aus:

$$\tilde{K}z \;=\; (I - \tilde{Q}\tilde{Q}^\star)y$$

Da $\tilde{Q}z = 0$, folgt daraus:

$$\begin{aligned} Kz &= y - \tilde{Q}\alpha \\ \Leftrightarrow \quad z &= K^{-1}y - K^{-1}\tilde{Q}\alpha \end{aligned}$$

Die Bedingung $\tilde{Q}^\star z = 0$ führt zu:

$$\alpha \;=\; (\tilde{Q}^\star K^{-1}\tilde{Q})^{-1}\tilde{Q}^\star K^{-1}y$$

Der Vektor $\hat{y} := K^{-1}y$ wird dabei durch Lösung von $K\hat{y} = y$ berechnet und ebenso wird $\hat{Q} := K^{-1}\tilde{Q}$ durch $K\hat{Q} = \tilde{Q}$ gelöst.

Die letzte Gleichung muss dabei nur einmal in einem Iterationsprozeß gelöst werden. Daher sind effektiv nur $i_s + k$ Operationen mit dem Vorkonditionierer für i_s Iterationen mit dem linearen Löser erforderlich. Die Matrix-Vektormultiplikation mit dem links-vorkonditionierenden Operator in einer Iteration mit dem Krylov-Löser erfordert nur eine Operation mit $\tilde{Q}^\star$ und $K^{-1}\tilde{Q}$, anstatt vier Multiplikationen mit dem Projektionsoperator $I - \tilde{Q}\tilde{Q}^\star$.

Offensichtliche Einsparungen werden erzielt, wenn man den Operator K für eine Reihe von aufeinanderfolgenden Eigenwertberechnungen beibehält. In diesem Fall kann man die Spalten von $\hat{Q}$ aus den vorherigen Schritten übernehmen.

4.7 Der vollständige Algorithmus

Der Jacobi-Davidson-Algorithmus beinhaltet die Möglichkeit eines Restarts, wenn die Dimension des gegenwärtigen Unterraums m_{max} überschreitet. Er berechnet k_{max} Eigenwerte nahe einem vorgegebenen Ziel τ in der komplexen Ebene. Hierbei muss man notwendigerweise ungenau sein, da die Eigenwerte einer gewöhnlichen nicht-hermiteschen Matrix in der komplexen Ebene nicht angeordnet sind. Welche Ritzwerte der Algorithmus als nahe Eigenwerte liefert, hängt unter anderem von den Winkeln der Eigenvektornäherungen zu den zugehörigen Eigenvektoren ab.

Gewöhnlich ist die Auswahl von Eigenwerten passend, wenn τ irgendwo aus dem äußeren Spektrum der Eigenwerte gewählt ist. Will man dagegen Eigenwerte von A aus dem Innern des Spektrums berechnen, so ist dieser Algorithmus wenig zufriedenstellend. In diesem Fall sollte man besser mit harmonischen Ritzwerten (siehe Kapitel 4.8) arbeiten.

Zur Anwendung des Algorithmus müssen eine Toleranzgrenze ϵ, das gewünschte Ziel τ und die Anzahl der Eigenpaare k_{max}, die um τ herum berechnet werden sollen, vorgegeben werden. m_{max} gibt die maximale Größe des Suchraums vor. Wird m_{max} überschritten, so findet ein Restart mit vorgegebener Dimension m_{min} statt. Nach Beendigung des Algorithmus erhält man k_{max} Eigenwerte nahe τ und die zugehörige Schurform $AQ = QR$, wobei Q eine $n \times k_{max}$ Orthogonal- und R eine $k_{max} \times k_{max}$ obere Dreiecksmatrix ist. Die Eigenwerte findet man dabei auf der Diagonalen von R. Die berechnete Schurform genügt der Gleichung $\|Aq_j - QRe_j\|_2 \leq j\epsilon$, wobei q_j die j-te Spalte von Q bezeichnet.

Es folgen nun einige Bemerkungen zu den einzelnen Schritten des Algorithmus, der in Abbildung 4.4 dargestellt ist:

1. Der neue Vektor t wird in Bezug auf den vorliegenden Suchraum orthonormiert. Dies kann zum Beispiel mit dem Verfahren von Gram-Schmidt erreicht werden wie im dargestellten Algorithmus. Eine andere Möglichkeit ist das iterative modifizierte Gram-Schmidt-Verfahren (siehe Kapitel 3.2), was die numerische Stabilität verbessert.
2. Es wird die letzte Spalte und die letzte Zeile der dicht besetzten Matrix $M := V^\star AV = V^\star V^A$ der Ordnung m berechnet. Dabei ist $V^A := AV$ und V eine $n \times m$ Matrix mit den Spalten v_j. Das gleiche gilt für V^A.
3. Die Schurzerlegung der $m \times m$ Matrix M kann durch einen Standardlöser für dichte Eigenwertprobleme berechnet werden. In diesem Algorithmus werden die Standard-Ritzwerte berechnet, weshalb sich der Algorithmus zur Berechnung von k_{max} äußeren Eigenwerten von A in der Nähe von τ eignet. Zur Berechnung von inneren Eigenwerten sollte man einen Algorithmus mit harmonischen Ritzwerten verwenden.

   In jedem Schritt muss die Schurform so sortiert werden, dass $T_{1,1}$ am nächsten an τ liegt. Eine Ausnahme stellt hierbei der Fall dar, dass $m \geq m_{max}$ ist, denn dann sind alle m_{max} führenden Diagonalelemente am nächsten bei τ, so dass in diesem Fall die Sortierung übersprungen werden kann.
4. Eine Schurvektornäherung wird dann akzeptiert, wenn die Norm des Residuums kleiner als ϵ ist. Das bedeutet, dass Ungenauigkeiten in der Größenordnung von ϵ in den berechneten Eigenwerten akzeptiert werden. Die Ungenauigkeiten der Schurvektoren (in den Winkeln) liegen ebenfalls in der Größenordnung $\mathcal{O}(\epsilon)$, falls der betreffende Eigenwert einfach ist und separiert von den anderen Eigenwerten liegt.

   Handelt es sich bei der Matrix um eine reelle Matrix, so sind die Eigenwerte entweder auch alle reell oder sie kommen immer konjugiert komplex vor. Wird also ein komplexer Eigenwert gefunden, so kennt man den konjugiert komplexen und kann dies ausnutzen, was den

Algorithmus effizienter macht.

5. Wird ein Ritzpaar akzeptiert, so wird mit der Suche nach dem nächsten Paar fortgefahren. Dabei bilden die verbleibenden Ritzvektoren die Basis für den neuen Suchraum.
6. Überschreitet die gerade aktuelle Dimension des Suchraums m_{max}, so wird ein Restart durchgeführt. Dabei werden die m_{min} Ritzvektoren, deren zugehöriger Ritzwert am nächsten bei τ liegt, als Spannvektoren des neuen Suchraums verwendet.
7. In der Matrix Q befinden sich die berechneten Schurvektoren. Die Matrix $\tilde{Q}$ besteht aus der Matrix Q, erweitert um den gerade aktuellen Schurvektor u. So erhält man eine kompaktere Schreibweise.

   Die zu lösende Korrekturgleichung ist äquivalent zu

   $$P_m(I - Q_{k-1}Q_{k-1}^\star)(A - \theta_j^{(m)}I)(I - Q_{k-1}Q_{k-1}^\star)P_m t_j^{(m)} = -r_j^{(m)}$$

   Die neue Korrektur t muss dabei sowohl orthogonal zu den Spalten von Q als auch zu u sein.

   Die Korrekturgleichung kann mit jedem passenden Verfahren gelöst werden, zum Beispiel mit vorkonditionierten Krylov-Verfahren, die für unsymmetrische Systeme gedacht sind (siehe hierzu [11] oder [16]). Es muss sichergestellt werden, dass der Startvektor t_0 für den iterativen Löser die Orthogonalitätsbedingung $\tilde{Q}^\star t_0 = 0$ erfüllt.

   Die Korrekturgleichung muss dabei nicht sehr genau gelöst werden. Es ist hier sinnvoll, wie es oft für inexakte Newton-Verfahren genutzt wird, die Genauigkeit mit jedem Iterationsschritt des Jacobi-Davidson-Verfahrens zu erhöhen. Man könnte die Korrekturgleichung beispielsweise im j-ten Iterationsschritt mit einer Residuumreduktion von 2^{-j} lösen. Findet man einen Schurvektor, so wird j wieder auf Null zurückgesetzt. Diese Strategie macht Sinn, da man das Jacobi-Davidson-Verfahren auch als ein (inexaktes) Newton-Verfahren bezeichnen kann.

   In den ersten Schritten sollte man die Korrekturgleichung nicht exakt lösen, da der genäherte Eigenwert θ noch sehr ungenau sein wird. Hier kann es sogar effektiver sein, θ vorübergehend durch τ zu ersetzen oder $t = -r$ für die Erweiterung des Suchraums zu benutzen.

$t = v_0, k = 0, m = 0, Q = [\,], R = [\,]$
while $k < k_{max}$
(1) **for** $i = 1, .., m$
$t = t - (v_i^\star t) v_i$
end for
$m = m + 1$, $v_m = \frac{t}{\|t\|_2}$, $v_m^A = A v_m$
(2) **for** $i = 1, .., m-1$
$M(i,m) = v_i^\star v_m^A$, $M(m,i) = v_m^\star v_i^A$
end for
$M(m,m) = v_m^\star v_m^A$
(3) Schurzerlegung $M = STS^\star$, mit S orthogonal und
T obere Dreiecksmatrix, mit $|T_{i,i} - \tau| \leq |T_{i+1,i+1} - \tau|$
$x = Sy, u = Vx_1, u^A = V^A x_1, \theta = T_{1,1}, r = u^A - \theta u, \tilde{a} = Q^\star r, \tilde{r} = r - Q\tilde{a}$
(4) **while** $\|\tilde{r}\|_2 \leq \epsilon$
$R = \begin{pmatrix} R & \tilde{a} \\ 0 & \theta \end{pmatrix}$
$Q = [Q, u]$, $k = k + 1$
if $k = k_{max}$ **then**
stop
end if
$m = m - 1$
(5) **for** $i = 1, .., m$
$v_i = V x_{i+1}$, $v_i^A = V_{x_{i+1}}^A$, $x_i = e_i$
end for
M = unterer $m \times m$ Block von T
$u = v_1$, $\theta = M_{1,1}$, $r = v_1^A - \theta u$, $\tilde{a} = Q^\star r$, $\tilde{r} = r - Q\tilde{a}$
end while
(6) **if** $m \geq m_{max}$ **then**
for $i = 2, .., m_{min}$
$v_i = V x_i$, $v_i^A = V^A x_i$
end for
M = führender $m_{min} \times m_{min}$ Block von T
$v_1 = u$, $v_1^A = u^A$, $m = m_{min}$
end if
$\tilde{Q} = [Q, u]$
(7) Löse $t(\perp \tilde{Q})$ näherungsweise über : $(I - \tilde{Q}\tilde{Q}^\star)(A - \theta I)(I - \tilde{Q}\tilde{Q}^\star)t = -\tilde{r}$
end while

Abbildung 4.4: Jacobi-Davidson-Verfahren zur Berechnung von k_{max} äußeren Eigenwerten (JDQR)

4.8 Harmonische Ritzwerte

Wie schon in den vorangegangenen Kapiteln erläutert, generiert das Jacobi-Davidson-Verfahren Basisvektoren $v_1, ..., v_m$ für einen Unterraum $\mathcal{V}_m$. Für die Projektion der Matrix A in den Unterraum müssen die Vektoren $w_j := Av_j$ berechnet werden. Die Inversen der Ritzwerte der Matrix A^{-1} in Bezug auf den Unterraum, der durch die Vektoren w_j aufgespannt wird, nennt man ***harmonische Ritzwerte*** der Matrix A bezüglich des linearen Teilraums $\mathcal{W}_m$ (siehe auch 2.4 auf Seite 11). Die Berechnung der Inversen möchte man gerne vermeiden, da sie viel Rechenaufwand erfordert. Dies ist möglich, wenn man $\mathcal{W}_m := A\mathcal{V}_m$ als Orthonormalbasis aufbaut. Jetzt sind allerdings nicht mehr die harmonischen Ritzvektoren $\tilde{u} \in \mathcal{V}_m$ orthonormal, sondern $\hat{u} := A\tilde{u} \in \mathcal{W}_m$, das heißt ein harmonisches Ritzpaar $(\tilde{\theta}_j^{(m)}, \tilde{u}_j^{(m)})$ erfüllt die Bedingung

$$
\begin{aligned}
& A\tilde{u}_j^{(m)} - \tilde{\theta}_j^{(m)}\tilde{u}_j^{(m)} \perp \mathcal{W}_m := \text{span}(w_1, ..., w_m) \quad \text{mit} \quad \tilde{u}_j^{(m)} \in \mathcal{V}_m, \tilde{u}_j^{(m)} \neq 0 \\
\Leftrightarrow \quad & W_m^\star A\tilde{u}_j^{(m)} - \tilde{\theta}_j^{(m)} W_m^\star \tilde{u}_j^{(m)} = 0 \quad \text{mit} \quad \tilde{u}_j^{(m)} := V_m s_j^{(m)} \\
\Leftrightarrow \quad & W_m^\star A V_m s_j^{(m)} - \tilde{\theta}_j^{(m)} W_m^\star V_m s_j^{(m)} = 0 \\
\Leftrightarrow \quad & (W_m^\star V_m)^{-1} W_m^\star A V_m s_j^{(m)} - \tilde{\theta}_j^{(m)} s_j^{(m)} = 0
\end{aligned}
$$

Dies beinhaltet, dass die harmonischen Ritzpaare Eigenwerte des verallgemeinerten Eigenwertproblems $(W_m^\star A V_m, W_m^\star V_m)$ sind beziehungsweise die Eigenwerte der $(m \times m)$-Matrix $H_m := (W_m^\star V_m)^{-1} W_m^\star A V_m$ die harmonischen Ritzwerte von A sind (siehe [4]).

Die äußeren Standard-Ritzwerte konvergieren gewöhnlich gegen die äußeren Eigenwerte von A. Dementsprechend konvergieren die inneren harmonischen Ritzwerte der geshifteten Matrix $A - \tau I$ gewöhnlich gegen Eigenwerte $\lambda \neq \tau$, die am nächsten bei dem Shift τ liegen. Glücklicherweise stimmen der Suchraum $\mathcal{V}_m$ der geshifteten und der ungeshifteten Matrix überein, wodurch die Berechnung von harmonischen Ritzpaaren vereinfacht wird. Aus Stabilitätsgründen konstruiert man $\mathcal{V}_m$ und $\mathcal{W}_m$ so, dass sie beide näherungsweise orthonormal sind; das heißt, dass $\mathcal{W}_m$ so konstruiert wird, dass $(A - \tau I)V_m = W_m M_m^A$, wobei M_m^A eine $m \times m$ obere Dreiecksmatrix ist.

Die harmonischen Ritzvektoren beinhalten meist mehr Informationen als die harmonischen Ritzwerte. Besonders dann, wenn der genäherte Eigenvektor einen kleinen Winkel mit dem Eigenvektor bildet und das Ziel τ sehr nahe am zugehörigen Eigenwert liegt, ist der Ritzwert nicht sehr gut. Der Grund dafür ist, dass ein harmonisches Eigenpaar $(\tilde{\theta}_j^{(m)}, \tilde{u}_j^{(m)})$ zu dem Residuum

$$r = A\tilde{u}_j^{(m)} - \tilde{\theta}_j^{(m)}\tilde{u}_j^{(m)}$$

führt und es gilt :

$$r \perp (A - \tau I)\tilde{u}_j^{(m)}$$

Wenn nun sowohl $\tilde{u}_j^{(m)}$ als auch τ gute Approximationen für ein Eigenpaar von A sind, so kann $\tilde{\theta}_j^{(m)}$ nur eine schlechte Näherung werden. Deshalb hat man Restart-Strategien entwickelt, die die Rayleigh-Quotienten mit den harmonischen Ritzwerten verknüpfen.

Es ist hier sinnvoller, eine Schurzerlegung durchzuführen statt einer Eigenwertzerlegung. Dabei werden zuerst Vektoren $q_1, ..., q_k$ berechnet, so dass $AQ_k = Q_k R_k$ ist, wobei $Q_k^\star Q_k = I_k$ gilt und R_k eine $k \times k$ obere Dreiecksmatrix ist. Die Diagonalelemente von R_k bilden die Eigenwerte von A. Die zugehörigen Eigenvektoren von A können dann über Q_k und R_k berechnet werden.

Ein Problem beim Jacobi-Davidson-Verfahren mit harmonischen Ritzwerten besteht darin, dass der Suchraum $A - \tau I$ schlecht konditioniert ist, falls τ nahe an einem Eigenwert der Matrix A liegt.

Der Algorithmus hierzu sieht folgendermaßen aus (siehe hierzu auch [2]):

$t =$ random$, k = 0, m = 0, Q = [\,], R = [\,]$;
while $k < k_{max}$
$w = (A - \tau I)t$
(1) **for** $i = 1, .., m$
$\gamma = w_i^\star w$, $w = w - \gamma \star w_i$, $t = t - \gamma v_i$
end for
$m = m + 1$, $w_m = \frac{w}{\|w\|_2}$, $v_m = \frac{t}{\|w\|_2}$
(2) **for** $i = 1, .., m - 1$
$M_{i,m} = w_i^\star v_m$, $M_{m,i} = w_m^\star v_i$
end for
$M_{m,m} = w_m^\star v_m$
(3) Schurzerlegung $M = STS^\star$, mit S orthogonal und
T obere Dreiecksmatrix, mit $|T_{i,i} - \tau| \leq |T_{i+1,i+1} - \tau|$
$x = Sy$, $\tilde{u} = VS_1$, $\mu = \|\tilde{u}\|_2$,$u = \frac{\tilde{u}}{\mu^2}$, $\theta = \frac{T_{1,1}}{\mu^2}$, $\tilde{w} = WS_1$,$r = \frac{\tilde{w}}{\mu} - \theta u$,
$\tilde{a} = Q^\star r$, $\tilde{r} = r - Q\tilde{a}$
(4) **while** $\|\tilde{r}\|_2 \leq \epsilon$
$R = \begin{pmatrix} R & \tilde{a} \\ 0 & \theta + \tau \end{pmatrix}$
$Q = [Q, u]$, $k = k + 1$
if $k = k_{max}$ **then**
stop
end if
$m = m - 1$
(5) **for** $i = 1, .., m$
$v_i = Vx_{i+1}$, $w_i = Wx_{i+1}$, $x_i = e_i$
end for
M = unterer $m \times m$ Block von T
$\mu = \|v_1\|_2$, $\theta = \frac{T_{2,2}}{\mu^2}$, $u = \frac{v_1}{\mu}$, $r = \frac{w_1}{\mu} - \theta u$, $\tilde{a} = Q^\star r$, $\tilde{r} = r - Q\tilde{a}$
end while
(6) **if** $m \geq m_{max}$ **then**
for $i = 2, .., m_{min}$
$v_i = VS_i$, $w_i = WS_i$
end for
M = führender $m_{min} \times m_{min}$ Block von T
$v_1 = \tilde{u}$, $w_1 = \tilde{w}$, $m = m_{min}$
end if
$\tilde{Q} = [Q, u]$
(7) löse $t(\perp \tilde{Q})$ näherungsweise über : $(I - \tilde{Q}\tilde{Q}^\star)(A - (\theta + \tau)I)(I - \tilde{Q}\tilde{Q}^\star)t = -\tilde{r}$
end while

Abbildung 4.5: Jacobi-Davidson-Verfahren zur Berechnung von k_{max} inneren Eigenwerten über harmonische Ritzwerte

Kapitel 5

Programmiertechniken in MATLAB

5.1 Einleitung

In diesem Kapitel werden die Programmiertechniken erläutert, die zur Erstellung der Bedienoberfläche benötigt wurden. Um das Programmpaket übersichtlich zu gestalten, wurden der Quellcode für die Benutzeroberfläche und die Algorithmen voneinander getrennt in unterschiedlichen Dateien, sogenannten ***m-Files***, gespeichert. ***m-Files*** sind ASCII-Files und werden mit einem Texteditor geschrieben. Sobald sie im Kommandofenster aufgerufen werden, führt sie der MATLAB-Interpreter wie ein Programm aus. Diese Dateien sind in verschiedene Funktionen unterteilt, die entweder aufgerufen werden können oder als ***events*** ausgelöst werden, wenn der Benutzer bestimmte Aktionen auf der Oberfläche ausführt. Des Weiteren wird ein Einblick in die Oberflächenprogrammierung von MATLAB gegeben.

5.2 Funktionen

Wie in jeder höheren Programmiersprache ist es auch in MATLAB möglich, Sequenzen von Befehlen in ***Funktionen*** zusammenzufassen. Funktionen sind, im Gegensatz zu Skripten, ***m-Files***, die Eingabewerte akzeptieren und Ausgabeargumente zurückgeben. Skripte hingegen sind nur Dateien, die eine Sequenz von MATLAB-Befehlen enthalten. Sie enthalten keine lokalen Variablen und haben keinen eigenen Arbeitsbereich. Jede Funktion hingegen hat ihren eigenen Arbeitsbereich, der getrennt ist vom Arbeitsbereich des MATLAB- Kommandofensters. Der Name des ***m-Files*** und der Name der Funktion sollten dabei übereinstimmen. Funktionen werden wie folgt definiert:

```
function [ out1, out2, ... ] = funcname(in1, in2, ...)
```

Abbildung 5.1: Syntax einer Funktion

Alle Variablen innerhalb einer Funktion sind lokal, außer sie werden als global vereinbart. Lokale Variablen müssen nicht deklariert werden, globale Variablen dagegen schon. Globale Variablen sind in allen Funktionen bekannt, in denen sie deklariert werden.

Funktionsnamen müssen mit einem Buchstaben beginnen. Der Funktionsname sollte mit dem Namen des ***m-Files*** bis auf die Endung ***.m*** übereinstimmen, da MATLAB beim Aufruf nach diesem Dateinamen sucht, das heißt MATLAB sucht nicht nach einer Funktion mit diesem Namen, sondern nach einer Datei. Existiert keine Datei mit diesem Namen, so findet MATLAB die Funktion nicht. Der Funktionsname spielt für MATLAB keine Rolle, wichtig ist der Dateiname.

Enthält ein ***m-File*** mehrere Funktionen, so sind alle Funktionen außer der ersten Unterfunktionen. Diese sind nur innerhalb der Datei bekannt und können von Funktionen in anderen Dateien nicht aufgerufen werden. Wird innerhalb eines ***m-Files*** eine Funktion aufgerufen, so sucht MATLAB zuerst nach Unterfunktionen und erst dann nach ***built-in***-Funktionen oder Funktionen innerhalb einer anderen Datei. So ist es also möglich, bereits bestehende Funktionen zu überschreiben.

Funktionen können mit einem ***end***-Statement abgeschlossen werden. Dies ist nur dann erforderlich, wenn es sich um geschachtelte Funktionen handelt. Schließt man eine Funktion mit einem ***end***-Statement ab, so müssen auch alle anderen Funktionen innerhalb des ***m-Files*** mit dem ***end***-Statement abgeschlossen werden.

Beispiel einer MATLAB-Funktion:

```
function [ mean,stdev ] = stat(x)
                  n = length(x);
                  mean = sum(x)/n;
                  stdev = sqrt(sum((x-mean).^2/n));
```

Abbildung 5.2: Beispiel einer Funktion

Hierbei bedeutet der Punkt, dass bei der Berechnung von **stdev** elementweise quadriert wird. Die ***built-in***-Funktion **sum** bildet die Summe über alle Elemente des Eingabeargumentes x.
Ausgabeargumente sind der Mittelwert und die Standardabweichung.
Erklärungen zu diesen und weiteren Funktionen findet man sowohl in den Hilfeseiten als auch in der MATLAB Online-Dokumentation (siehe [18]).

5.3 Zuweisungen

Zuweisungen von Skalaren sind in MATLAB mit dem "="-Operator möglich. In MATLAB sind aber, im Gegensatz zum Beispiel zur Programmiersprache C, auch Array-Zuweisungen möglich, die sehr effizient sind. Beispielsweise können Matrizen der gleichen Größe in einer Anweisung addiert werden, wofür man in der Programmiersprache C eine doppelte Schleife benötigt.

Hierzu ein kurzes Beispiel :

Soll von je 4 aneinander angrenzenden Matrixelementen der Mittelwert gebildet werden und dies über die gesamte Matrix, so benötigt man in C eine doppelt verschachtelte Schleife, um über alle Matrixelemente zu gehen. In MATLAB ist dies nicht notwendig. Man kann hier einen Indexbereich angeben, der aus der Matrix extrahiert und mit anderen Matrizen der gleichen Größe addiert werden soll.

Beispielsweise liefert die Matrix:

$$\mathrm{A} = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 4 & 5 & 6 & 7 \\ 6 & 7 & 8 & 9 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

den Output :

$$\text{mean} = \begin{pmatrix} 3.5 & 4.5 & 5.5 \\ 5.5 & 6.5 & 7.5 \\ 7.5 & 8.5 & 9.5 \end{pmatrix}$$

Der Quelltext in C im Vergleich zu MATLAB sieht dann folgendermaßen aus, wobei *gr_x* die Anzahl der Zeilen und *gr_y* die Anzahl der Spalten der Matrix *A* bezeichnet:

```
for(i=1; i<gr_x; ++i)
    for(j=1; j<gr_y; ++j)
        mean[i-1][j-1] = ( A[i-1][j] + A[i-1][j-1]
                         + A[i][j]   + A[i][j-1] ) / 4;
```

Abbildung 5.3: Zuweisung in C

```
[gr_x, gr_y] = size(A);
i = 2:gr_x;
j = 2:gr_y;
mean = ( A(i-1,j) + A(i-1,j-1) +
          A(i,j)   + A(i,j-1) ) / 4;
```

Abbildung 5.4: Zuweisung in MATLAB

5.4 Oberflächenprogrammierung

Für die Programmierung von graphischen Benutzeroberflächen (**G**raphical **U**ser **I**nterface) steht in MATLAB die Oberfläche ***guide*** zur Verfügung. Beim Abspeichern wird sowohl ein ***fig-File*** als auch ein ***m-File*** erzeugt. Das ***fig-File*** enthält dabei die vollständige Beschreibung der Benutzeroberfläche und ihres Layouts. Das ***m-File*** enthält den Code, der der Kontrolle der GUI dient sowie alle ***Callbacks***. Ein ***Callback*** ist eine Funktion, die dann ausgeführt wird, wenn der Benutzer einen Teil der Benutzeroberfläche aktiviert, zum Beispiel durch Anklicken eines Buttons. Der Name des ***Callbacks*** wird durch die Eigenschaft "Tag" bestimmt, die jedes Element besitzt. Wird beispielsweise ein Button mit dem "Tag" ***start*** gedrückt, so wird der ***Callback***

function start_Callback(hObject, eventdata, handles)

aufgerufen.

- **hObject** enthält dabei das Handle (eindeutige Kennung), von dem es aufgerufen wurde, in diesem Fall des Buttons.
- **eventdata** wird bisher nicht benötigt. Dies wird sich allerdings bei neuen Versionen von MATLAB ändern.
- **handles** ist eine Struktur mit Handles und den Benutzerdaten.

Ein Handle wird beispielsweise benötigt, wenn man für ein Objekt auf der Oberfläche die Sichtbarkeit ändern möchte. Befindet man sich im entsprechenden **Callback** des Objektes, so enthält der Übergabeparameter **hObject** diese Kennung. Befindet man sich jedoch in einer anderen Funktion oder einem anderen **Callback**, so muss diese Kennung erst mit dem Kommando

hand = findobj('Tag', 'vorkonditionierer');

bestimmt werden, um danach dessen Sichtbarkeit mit dem Kommando

set(hand, 'Visible', 'on'); beziehungsweise **set(hand, 'Visible', 'off');**

zu ändern. Bei diesem Beispiel würde somit die Sichtbarkeit des Objektes, dessen "Tag" auf "vorkonditionierer" gesetzt ist, verändert.

Callbacks werden auch beim Erzeugen und Zerstören des Objekts aufgerufen. Elemente wie zum Beispiel Buttons haben hierfür eine eigene Funktion

function start_CreateFcn(hObject, eventdata, handles)

Die Art und Anzahl der Übergabeparameter sind bei allen ***Callbacks*** identisch.

5.5 Einbinden von in C oder Fortran geschriebenen Funktionen

Es ist in MATLAB möglich, externe, in C oder Fortran geschriebene Funktionen einzubinden. Dazu muss man sogenannte ***MEX-Files*** schreiben. Dies sind dynamisch gelinkte Unterprogramme, die der MATLAB-Interpreter automatisch laden und ausführen kann. Ein Vorteil dieser Methode ist zum Beispiel, dass große bereits bestehende C- oder Fortran-Programme aus MATLAB heraus aufgerufen werden können, ohne dass sie vorher als ***m-File*** neu geschrieben werden müssen. Des Weiteren können ***Bottleneck-Berechnungen*** (gewöhnlich ***for-Schleifen***), die in MATLAB nicht schnell genug sind, aus Effizienzgründen in C oder Fortran programmiert werden. ***MEX-Files*** sind Unterprogramme, die durch C- oder Fortran-Quellcode entstehen. Sie verhalten sich wie ***m-Files*** oder ***built-in***-Funktionen. Im Gegensatz zu ***m-Files***, die eine plattform-unabhängige Endung ***.m*** besitzen, ist die Endung bei ***MEX-Files*** plattform-spezifisch; auf ***LINUX***-Systemen haben sie die Endung ***.mexglx***, auf ***Windows***-Systemen die Endung ***.dll***. ***MEX-Files*** können genauso wie ***m-Files*** aufgerufen werden.
Wird in MATLAB eine Funktion aufgerufen, so schaut der MATLAB-Interpreter alle Verzeichnisse durch, die im MATLAB-Suchpfad eingetragen sind. Er durchsucht alle Verzeichnisse nach Dateien, die den Funktionsnamen mit der Endung ***.m*** oder der entsprechenden plattform-spezifischen Endung für ***MEX-Files*** haben. Bei der ersten Übereinstimmung nimmt MATLAB diese Datei, lädt die Funktion und führt sie aus. Existiert sowohl ein ***MEX-File*** als auch ein ***m-File*** im gleichen Verzeichnis, so wird das ***MEX-File*** genommen.

Die übersetzten ***MEX-Files*** müssen sich entweder im Verzeichnis des MATLAB-Pfades befinden oder in dem Verzeichnis, in dem MATLAB ausgeführt wird. Befindet sich in beiden Pfaden die gewünschte Funktion, so wird die Funktion im Verzeichnis, in dem MATLAB ausgeführt wird, bevorzugt. Neue Pfade können mit der Funktion ***addpath*** zum MATLAB-Pfad hinzugefügt werden.

Um ***MEX-Files*** verstehen zu können, muss man wissen, dass es in MATLAB nur einen einzigen Datentyp gibt, nämlich das MATLAB-Feld. Alle MATLAB-Variablen wie Skalare, Vektoren, Matrizen, Zeichenketten, Cell-Arrays (Feld, dessen Zellen gemischte Felder enthalten können) und Strukturen werden in MATLAB als Feld abgespeichert. In C deklariert man MATLAB-Felder mit ***mxArray***, in Fortran werden sie als Zeiger auf Integer empfangen. Die Struktur ***mxArray*** in C enthält unter anderem den Typ des Feldes, die Dimensionen, die enthaltenen Daten, bei numerischen Feldern, ob die Einträge reell oder komplex sind, bei dünn besetzten Matrizen die Indizes und die Nicht-Null-Elemente und bei einer Struktur die Anzahl der Felder und die Feldnamen. Alle Daten in MATLAB werden wie in Fortran spaltenweise abgespeichert, da MATLAB ursprünglich in Fortran geschrieben wurde. Komplexe Matrizen werden als zwei Vektoren gespeichert, einer enhält dabei die Realteile, der andere die Imaginärteile. MATLAB unterstützt neben komplexen Matrizen

auch einfachgenaue Gleitkommazahlen, 32-Bit Ganzzahlen, beide ohne und mit Vorzeichen. Zeichenketten werden wie 16-Bit Ganzzahlen ohne Vorzeichen abgespeichert. Im Gegensatz zu C sind Zeichenketten in MATLAB nicht mit einem '\0' abgeschlossen. ***MEX-Files*** in C unterstützen alle MATLAB-Datentypen, wohingegen in Fortran nur doppeltgenaue $n \times m$ Matrizen und Zeichenketten unterstützt werden.

Der Quellcode eines ***MEX-Files*** besteht aus zwei unterschiedlichen Teilen:

- der eigentlichen Routine
- einer sogenannten ***Gateway-Routine***, die ein Interface zwischen der eigentlichen Routine und MATLAB darstellt.

Der Eintrittspunkt für MATLAB ist die **Gateway-Routine**, die den Namen **mexFunction** tragen muss. Sie erhält als Parameter **prhs**, **nrhs**, **plhs** und **nlhs**. **prhs** ist dabei ein Feld, das die Eingabeargumente enthält, **plhs** entsprechend ein Feld mit den Ausgabeargumenten. **nrhs** enthält die Anzahl der Eingabeargumente, **lrhs** entsprechend die Anzahl der Ausgabeargumente. Die **Gateway-Routine** ruft die eigentliche Routine auf.

Damit der Eintrittspunkt und die Interface-Routinen richtig deklariert sind, muss in C die Headerdatei **mex.h** eingebunden werden. Die **Gateway-Routine** muss in C folgende Form haben:

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
```

Abbildung 5.5: Deklaration einer Gateway-Routine in C

In Fortran sieht sie ähnlich aus:

```
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      integer plhs(*), prhs(*)
      integer nlhs, nrhs
```

Abbildung 5.6: Deklaration einer Gateway-Routine in Fortran

Ruft man ein ***MEX-File***, zum Beispiel zur Multiplikation zweier Zahlen, aus MATLAB mit dem Kommando

$$x = \text{mult}(y, z)$$

auf, so ruft der MATLAB-Interpreter die Funktion **mexFunction** in C mit den Argumenten

$$\text{nlhs} = 1$$

$$\text{nrhs} = 2$$

auf, wobei der Zeiger **plhs** auf einen noch uninitialisierten Bereich, **prhs(0)** auf y und **prhs(1)** auf z zeigt, das heißt **plhs** ist ein Feld mit einem Element, das den Null-Pointer enthält und **prhs** ein Feld mit zwei Elementen, wobei das erste Element ein Zeiger auf ein **mxArray** mit Namen y und das zweite Element ein Zeiger auf ein **mxArray** mit Namen z ist.

Ein Beispiel für ein ***C-MEX-File*** ist:

```
#include "mex.h"

void orthogonalisiere(double * I, int size_I,
          int flag_basis_I, double * status) {

    /* eigentliche Routine */
}

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])  {

  int m_I, n_I, size_I;
  double * I;
  int flag_basis_I;
  double * status;  /* als Rueckgabewert */

  /* Create a pointer to the input matrix I */
  I = mxGetPr(prhs[0]);

  /* Get the dimensions of the matrix input I */
  m_I = mxGetM(prhs[0]);
  n_I = mxGetN(prhs[0]);
  size_I = m_I * n_I;

  /* Get the scalar input flag_basis_I */
  flag_basis_I = mxGetScalar(prhs[1]);

  /* Create matrix for the return argument. */
  /* Rueckgabewert : */
  plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);

  status = mxGetPr(plhs[0]);

  /* Call the C subroutine. */
  orthogonalisiere(I, size_I, flag_basis_I, status);
}
```

Abbildung 5.7: Beispiel eines MEX-Files in C

Kapitel 6

Funktionalität des Programmpaketes

6.1 Überblick

Das Programmpaket, das im Rahmen dieser Arbeit entstanden ist, gliedert sich in zwei wesentliche Teile, zum einen in die graphische Benutzeroberfläche und zum anderen in den Jacobi-Davidson-Algorithmus.

Die Datei ***umgebung.m*** enthält die Initialisierungsfunktionen und ***Callbacks*** der Benutzeroberfläche. Weitere Fenster, die bei Bedarf geöffnet werden, befinden sich in den Dateien ***start_button.m***, ***start_button_innen.m*** und ***ausgabe.m***. Startet man die Berechnung, so öffnet sich ein Fenster zur Eingabe der benötigten Vorgaben (Funktion ***start_button***, falls äußere Eigenwerte beziehungsweise ***start_button_innen***, falls innere Eigenwerte berechnet werden sollen).
Ist das Verfahren konvergent, so werden die gewünschten Eigenwerte sowie, falls das Jacobi-Davidson-Verfahren gewählt wurde, die Anzahl der benötigten Skalar- und Matrix-Vektorprodukte in einem Fenster angezeigt. Es wird ebenfalls eine Datei angelegt, in der sowohl Eigenwerte als auch Eigenvektoren im ASCII-Format abgespeichert werden (Funktion ***ausgabe***).
Die Dateien ***mmread.m*** und ***mmwrite.m*** enthalten I/O Funktionen, um Matrizen im Matrix-Market-Format einzulesen und wegzuschreiben.
Die Datei ***jada.m*** enthält den Jacobi-Davidson-Algorithmus für äußere Eigenwerte und ruft zur Sortierung der benötigten Schurform die Funktion ***schurtau*** aus der Datei ***schurtau.m*** auf. Für innere Eigenwerte existiert die Funktion ***jada_innen*** in der Datei ***jada_innen.m***.

Darüber hinaus werden Dateien für die externen Schnittstellen bereitgestellt, die im Kapitel 6.3 auf Seite 48 näher erläutert werden.

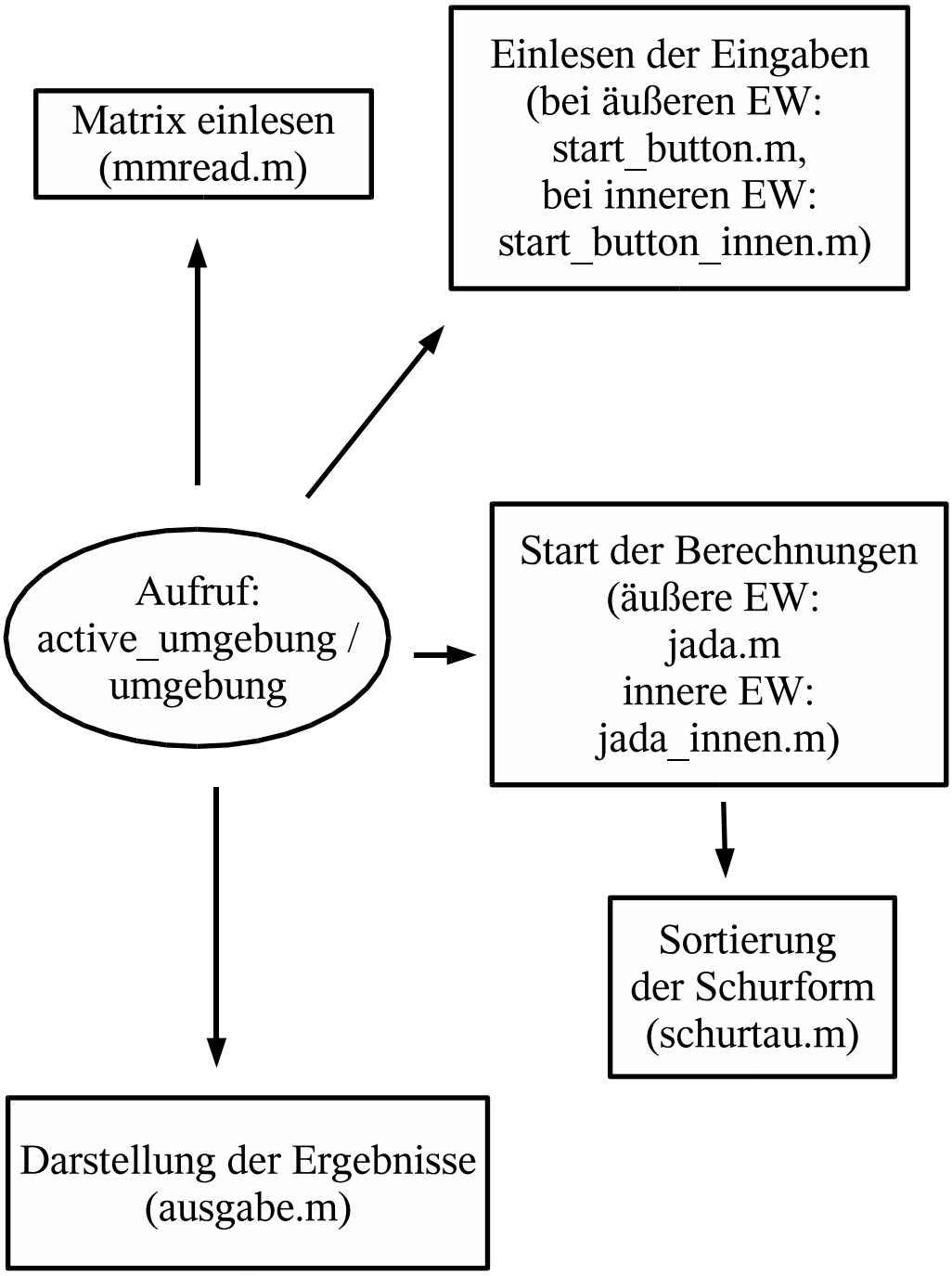

Abbildung 6.1: Dateien und ihre Abhängigkeiten

6.2 Die graphische Oberfläche

Gestartet wird die Benutzeroberfläche innerhalb des MATLAB-Kommandofensters mit dem Kommando

umgebung

Falls der Benutzer externe Routinen anbinden möchte, so muss sie mit dem Kommando

active_umgebung

gestartet werden, da hierbei gleichzeitig mit der Benutzeroberfläche die Sockets, die zur Kommunikation zwischen MATLAB und dem externen Programm erforderlich sind, gestartet werden.

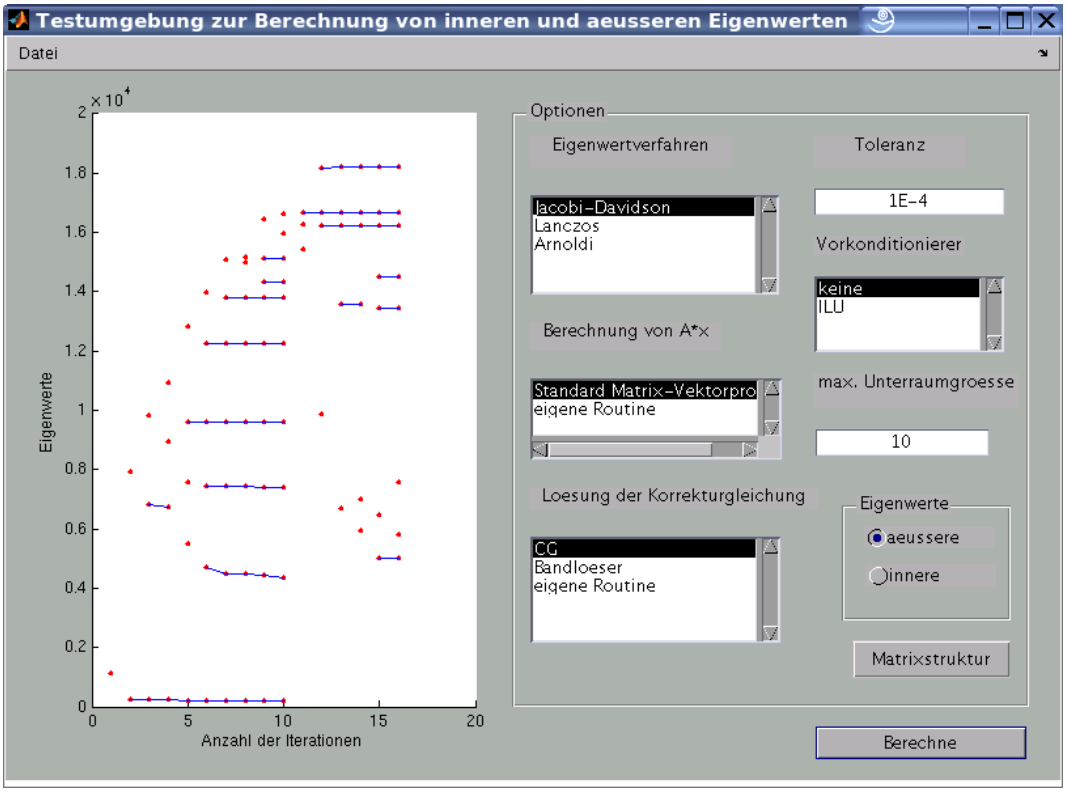

Abbildung 6.2: Benutzeroberfläche

Die Oberfläche gliedert sich in zwei Teile. Auf der linken Seite befindet sich ein Plot-Fenster, in dem man beim Jacobi-Davidson-Verfahren verfolgen kann, in welcher Iteration welche Eigenwertnäherung erreicht wurde. Die rechte Seite beinhaltet Optionen, die der Benutzer seinem Problem entsprechend einstellen kann. Mögliche Verfahren zur Eigenwertberechnung sind das Jacobi-Davidson-Verfahren, das Lanzcos-Verfahren für symmetrische Matrizen und das Arnoldi-Verfahren für unsymmetrische Matrizen. Zur Berechnung der vorkommenden Matrix-Vektorprodukte kann man entweder das Standard-Matrix-Vektorprodukt von MATLAB einsetzen oder eine eigene Routine anbinden. Bei der ersten Möglichkeit muss man zuvor eine Matrix in MATLAB einlesen. Bindet man eine eigene Routine zur Berechnung der Matrix-Vektorprodukte ein, so wird die Matrix außerhalb von MATLAB gespeichert. Dann muss nicht nur das Matrix-Vektorprodukt mit einer eigenen Routine berechnet werden, sondern auch das Skalarprodukt. Eine Funktion zur Initialisierung, zur Orthogonalisierung und eine Funktion für die Berechnung von Linearkombinationen sowie eine Funktion zur Lösung der Korrekturgleichung müssen in diesem Fall ebenfalls vom Benutzer bereitgestellt werden.

Wählt man als Verfahren den Jacobi-Davidson-Algorithmus und die Standardvariante für Matrix-Vektorprodukte von MATLAB aus, so muss noch ein Verfahren zur Lösung der Korrekturgleichung angegeben werden. Möglich sind hierbei CG-Verfahren, ein Bandlöser oder eigene Methoden. Entscheidet man sich hierbei für CG-Verfahren, so kann man noch einen Vorkonditionierer angeben, beispielsweise eine unvollständige LU-Zerlegung. Des Weiteren muss die maximale Unterraumgröße angegeben werden. Wird diese erreicht, so findet ein Restart statt.
Soll die Korrekturgleichung mit einem eigenen Verfahren gelöst werden, so muss hierfür eine Funktion vom Benutzer bereitgestellt werden (beim Lanczos- oder Arnoldi-Verfahren sind diese Eingaben überflüssig).

Eine Toleranzgrenze muss ebenfalls angegeben werden. Diese legt fest, welche Fehler in den Eigenwerten höchstens akzeptiert werden.

Verfahren	***Optionen***
Jacobi-Davidson-Verfahren	Standard-Matrix-Vektorprodukt CG-Löser ohne Vorkonditionierer Unterraumgröße Toleranz
Jacobi-Davidson-Verfahren	Standard-Matrix-Vektorprodukt CG-Löser ILU als Vorkonditionierer Unterraumgröße Toleranz
Jacobi-Davidson-Verfahren	Standard-Matrix-Vektorprodukt Bandlöser Unterraumgröße Toleranz
Jacobi-Davidson-Verfahren	Standard-Matrix-Vektorprodukt eigene Löser-Routine Unterraumgröße Toleranz
Jacobi-Davidson-Verfahren	eigene Routine für Matrix-Vektorprodukt eigene Löser-Routine Unterraumgröße Toleranz
Lanczos-Verfahren	Toleranz
Arnoldi-Verfahren	Toleranz

Tabelle 6.1: Mögliche Kombinationen

Mit den zwei Buttons im Panel **Eigenwerte** wird ausgewählt, ob man innere oder äußere Eigenwerte berechnen will. Je nach Auswahl wird entweder das Jacobi-Davidson-Verfahren mit Standard-Ritzwerten oder das Jacobi-Davidson-Verfahren mit harmonischen Ritzwerten aufgerufen.

Über das Menu ist es möglich, Matrizen einzulesen und abzuspeichern. Als Formate sind hierbei das ASCII-Format, das Binär-Format und das Matrix-Market-Format möglich. Dateien im Matrix-Market-Format müssen hierbei auf ***.mtx*** enden; Binär- und ASCII-Format brauchen keine spezielle Endung.
Abgespeichert werden kann ebenfalls in diesen drei Formaten. Matrix-Market-Dateien enden hierbei wieder auf ***.mtx***, ASCII-Dateien auf ***.ascii*** und Binärdateien auf ***.bin***.

Weiterhin ist es möglich, eine ausführbare Datei zu verwenden oder aber ein C- oder Fortran-Programm zu übersetzen und anschließend auszuführen. Kompiliert wird dabei unter Linux mit dem **Gnu C-Compiler gcc** oder mit dem **NAG Fortran-Compiler f90**. Unter Windows muss hierfür der **Borland C-Compiler** vorhanden sein. Fortran-Programme können hier bisher nicht kompiliert werden. Beim Kompilieren wird eine ausführbare Datei **prog.out** beziehungsweise **prog.exe** erzeugt, die anschließend ausgeführt wird.

Der Button ***Matrix*** zeigt sowohl die Besetzungsstruktur als auch die Werte der eingelesenen Matrix graphisch an.

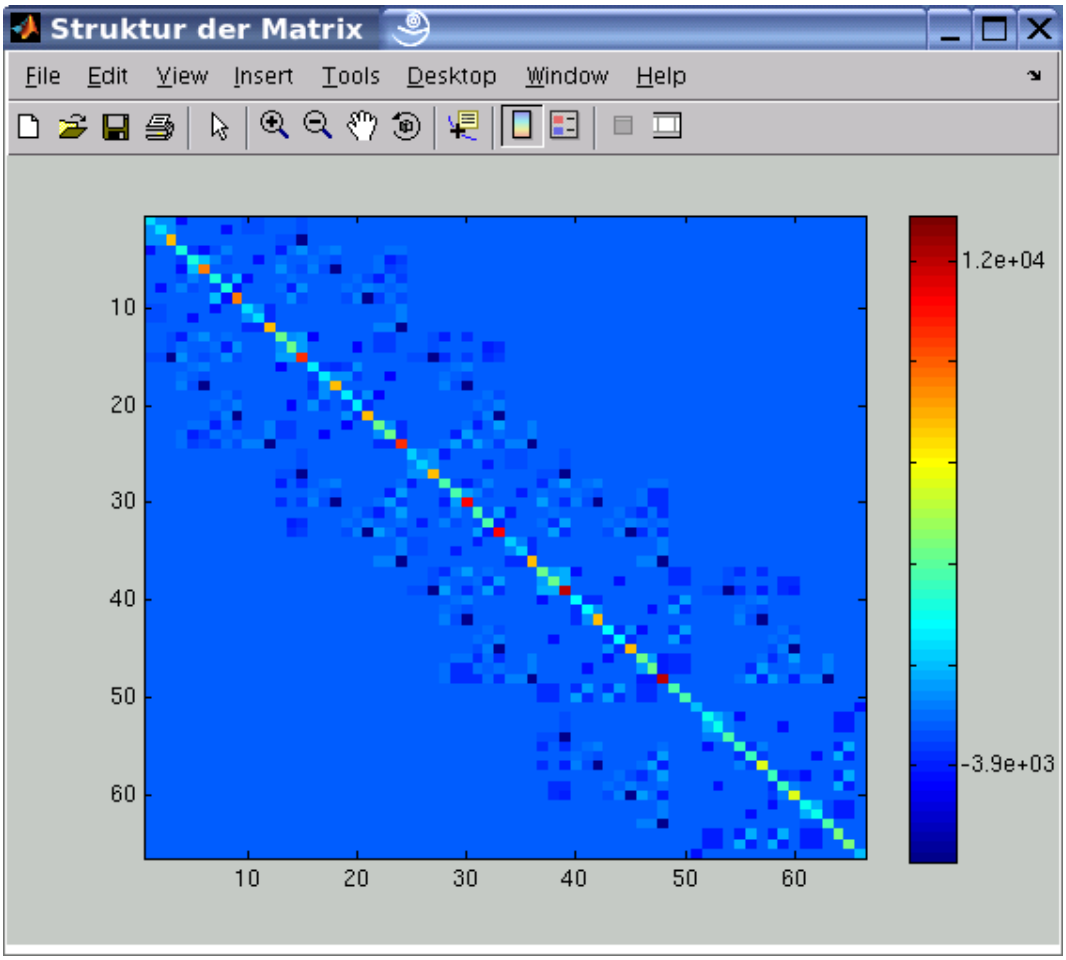

Abbildung 6.3: Beispielmatrix, siehe Anhang A

Der Button ***Start*** startet die Berechnung. Es wird dann ein weiteres Fenster geöffnet, in dem für die Berechnung notwendige Eingaben gemacht werden müssen.

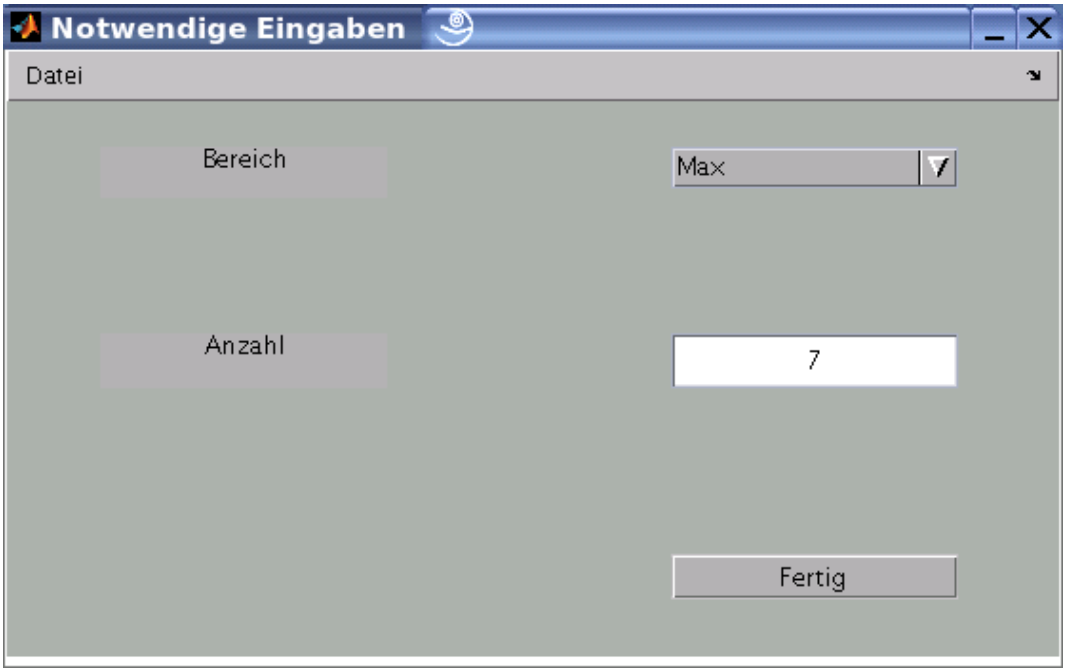

Abbildung 6.4: Notwendige Eingaben beim Jacobi-Davidson-Verfahren für äußere EW

Erstens muss der Bereich angegeben werden, in dem die Eigenwerte berechnet werden sollen. Möglich sind hierbei die betragsgrößten (**Max**), die betragskleinsten (**Min**) und die Eigenwerte um einen bestimmten Bereich. Werden drei Eigenwerte um den Bereich τ = 100 gewünscht, so werden die drei Eigenwerte λ berechnet, die den kleinsten Abstand von τ besitzen, das heißt bei denen $||\lambda - \tau||_2$ am kleinsten ist. Im komplexen Zahlenbereich bedeutet dies, dass innerhalb einer Kugel mit Mittelpunkt τ gesucht wird.
Darüber hinaus muss die Anzahl der gewünschten Eigenwerte eingegeben werden. Zur Bestätigung der Eingaben muss der Button ***Fertig*** gedrückt werden.

Beim Jacobi-Davidson-Verfahren für innere Eigenwerte muss immer ein Bereich angegeben werden. Die Auswahl ***Max*** oder ***Min*** ist hierbei nicht möglich, da es sich um innere Eigenwerte handelt. Der Aufbau dieses Fensters ist ähnlich. Auch hier muss zusätzlich zum Bereich die Anzahl der gewünschten Eigenwerte eingegeben werden.

Die Eigenwerte und Eigenvektoren werden dann mittels des gewünschten Verfahrens berechnet und es öffnet sich ein weiteres Fenster. Dieses zeigt die Eigenwerte an und hier sollte der Name einer Datei angegeben werden, in der sowohl die Eigenwerte als auch die Eigenvektoren abgespeichert werden.

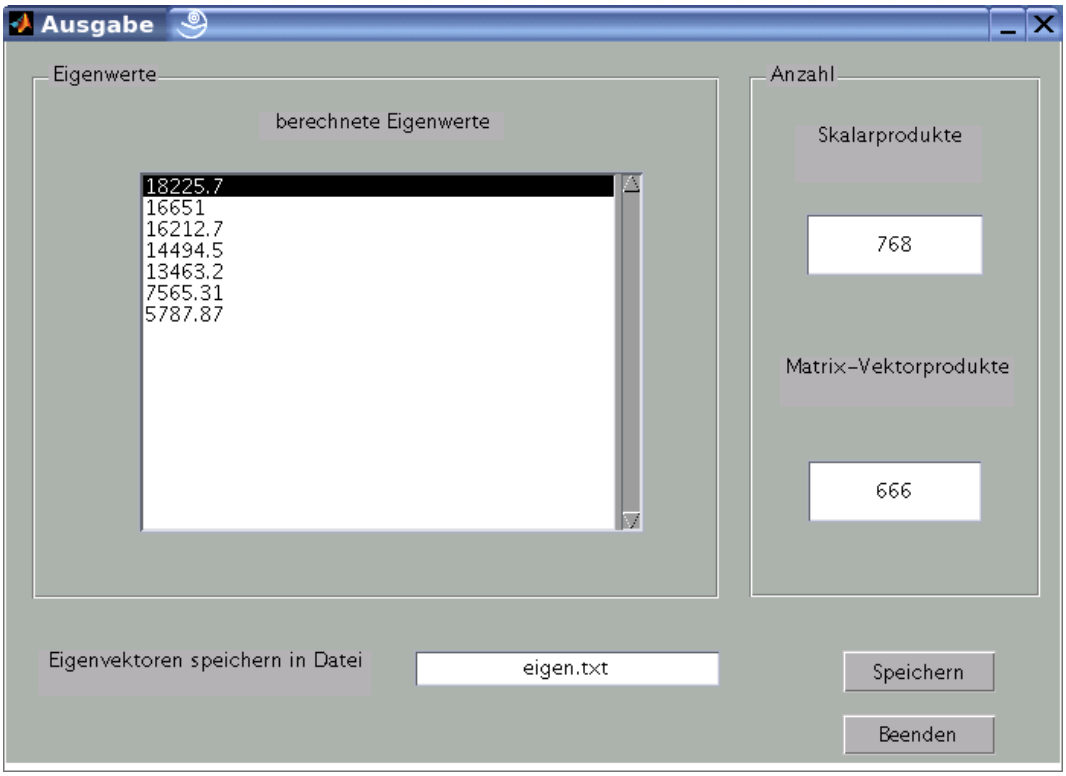

Abbildung 6.5: Fenster der Ausgabe

Wird der Button ***Beenden*** gedrückt, so werden weder Eigenwerte noch Eigenvektoren abgespeichert, sonst werden sowohl Eigenwerte als auch Eigenvektoren unter dem angezeigten Dateinamen (hier: *eigen.txt*) abgespeichert. Falls als Verfahren das Jacobi-Davidson-Verfahren ausgewählt wurde, wird zusätzlich angezeigt, wie viele Matrix-Vektor- und Skalarprodukte bei der Berechnung der Eigenwerte und Eigenvektoren benötigt wurden.

6.3 Schnittstellen

Es ist möglich, bestimmte Operationen nicht in MATLAB durchzuführen, sondern dafür externe Routinen zu nutzen. Man kann zum Beispiel angeben, dass zur Lösung der Korrekturgleichung kein CG-Verfahren und kein Bandlöser verwendet wird, sondern eine selbst geschriebene Routine. Diese Routine muss entweder in C oder in Fortran geschrieben sein. Einbinden externer Routinen ist nur über ***MEX-Files*** möglich (siehe Kapitel 4.5, Seite 39). Diese werden mit dem Kommando

mex [options] filename

im MATLAB-Kommandofenster kompiliert. Optionen kann man auch in einer Datei zusammenfassen und beim Kompilieren mit der Option **-f** angeben.
Wird keine Datei mit Optionen angegeben, so wird defaultmäßig unter Windows die Datei **mexopts.bat** und unter Linux die Datei **mexopts.sh** genommen, die sich im MATLAB-Unterverzeichnis **bin** befindet. Diese Datei enthält compilerspezifische Optionen zur Erzeugung eines ***MEX-Files***. Möchte man unter Linux Fortran90-Dateien kompilieren, so muss man den Compiler in dieser Datei ändern, da dieser zur Zeit noch standardmäßig auf **g77** gesetzt ist und somit keine Fortran90-Dateien kompiliert werden können. Um dies zu ermöglichen, wurde die Datei **mexopts_f90.sh** beigefügt (Verzeichnis **mexfiles**).

Erfolgt nur die Lösung der Korrekturgleichung extern, so wird die vollständige Speicherverwaltung innerhalb von MATLAB gehalten und es werden nur die Parameter an die externe C- oder Fortran-Routine übergeben, die zur Lösung der Korrekturgleichung nötig sind.

Hierfür existiert bereits ein ***MEX-File*** im Verzeichnis **mexfiles**, das die **Gateway-Routine** und einen leeren Funktionsrumpf enthält. Hier muss der Benutzer seine eigene Lösung implementieren und danach bei C-Dateien mit dem Kommando

mex korrektur_einzeln.c

beziehungsweise bei Fortran77-Dateien mit dem Kommando

mex korrektur_einzeln.f

kompilieren. Fortran90-Dateien werden mit dem Kommando

mex -f mexopts_f90.sh korrektur_einzeln.f90

kompiliert. Alle erzeugten **shared libraries** müssen entweder in das Hauptverzeichnis kopiert werden oder der Pfad muss mit der Funktion **path(path, pfadangabe)** zum MATLAB-Suchpfad hinzugefügt werden.

Die Übergabeparameter sowie der Name der Funktion sind dabei fest vorgegeben und können nicht verändert werden.

```
void korrektur_einzeln(double tau, double * res,
                       int groesse, double * Q, int size_Q,
                       double * A, int size_A, double * V,
                       int size_V, double * rueck)
```

Abbildung 6.6: Schnittstelle zur Lösung der Korrekturgleichung

mit

Parameter	***Datentyp***	***Bedeutung***
tau	double	Shift
res	double *	Residuum
groesse	int	Länge des Vektors **res**
Q	double **	Zeiger auf den Speicherbereich der iterierten Vektoren
size_Q	int	Anzahl der Spalten von Q
A	double **	Zeiger auf den Speicherbereich der Systemmatrix A
size_A	int	Anzahl der Spalten von A
V	double **	Zeiger auf den Speicherbereich der Basis V
size_V	int	Anzahl der Spalten von V
rueck	double	Rückgabewert Vektor, mit dem der Unterraum erweitert werden soll

Tabelle 6.2: Übergabeparameter der Funktion korrektur_einzeln

Der Aufruf dieser Funktion unter MATLAB sieht folgendermaßen aus:

```
rueck = korrektur_einzeln(double tau, double * res,
                          double * Q, double * A,
                          double * V)
```

Abbildung 6.7: Funktionsaufruf aus MATLAB

Weiterhin ist es möglich, die Systemmatrix A nicht in MATLAB einzulesen, sondern außerhalb von MATLAB zu halten. Ist dies der Fall, so müssen alle Informationen über das Eigenwertproblem außerhalb von MATLAB verwaltet werden. MATLAB kennt dann nur noch das projizierte Problem. Zusätzlich zu der Systemmatrix A müssen dann noch die Basisvektoren V und $W = AV$ sowie die iterierten Vektoren Q und einige nötige Hilfsvektoren extern gespeichert werden. Die Initialisierungsroutine muss dann nicht nur für die Matrix und die Startvektoren, falls vorhanden, Speicher anlegen, sondern auch für die benötigten Hilfsvektoren (alle Hilfsvektoren werden in einer Matrix zusammengefasst) und für die Eigenwertnäherungen. Man kann hier bereits Speicherplatz für die größtmögliche Basis anlegen, da man sonst den Speicher dynamisch erweitern muss, sobald ein neuer Vektor zur bereits bestehenden Basis hinzugenommen werden soll.

Werden nun externe Funktionen aus MATLAB heraus aufgerufen, so werden diesen Routinen Flags übergeben, die angeben, welche Matrix beziehungsweise welcher Vektor gerade bei der Berechnung benötigt wird.

Die Bedeutung der Flags ist den folgenden beiden Tabellen zu entnehmen.

Flag	***Matrix***
0	Systemmatrix A
1	Basis V
2	Basis W
3	Eigenwertnäherungen Q
4	Matrix der Hilfsvektoren

Tabelle 6.3: Extern gespeicherte Matrizen

Spalte	***Hilfsvektor***
1	u
2	u_A beziehungsweise u_tilde
3	r
4	r_tilde
5	a_tilde
6	Qa_tilde
7	w_tilde
8	W_1

Tabelle 6.4: Matrix mit den Hilfsvektoren

6.3.1 Kommunikation zwischen MATLAB und dem externen Programm

Die Kommunikation zwischen den externen Routinen und MATLAB wird über sogenannte **Sockets** gelöst. Hierbei existiert immer ein **Server-Socket**, das mit mehreren **Client-Sockets** kommunizieren kann. Wird der Server gestartet, so erstellt dieser zuerst ein Server-Socket und bindet dieses an eine freiwählbare Portnummer an. Über diese Portnummer werden dann Anfragen akzeptiert. Der Client muss ebenfalls ein Socket erstellen und verbindet sich mit diesem Socket an die Serveradresse, von der er Daten holen möchte. Die Portnummer muss dieselbe sein, an die das Serversocket angebunden ist. Nun können Daten zwischen dem Server und dem Client ausgetauscht werden.

Der Server wartet auf Anfragen vom Client. Schickt dieser eine Anfrage und der Server akzeptiert diese, so erstellt der Server ein Client-Socket und die Daten werden ausgetauscht. Anschließend trennt der Client die Verbindung und beendet sein Socket. Der Server schließt daraufhin sein Client-Socket und wartet auf eine neue Anfrage von einem Client. Das Server-Socket wird nicht geschlossen.
Diese Methode birgt eine gewisse Unsicherheit, da die gewählte Portnummer nicht exklusiv für den Server und den Client reserviert ist. Theoretisch ist es möglich, dass sich ein weiterer Client mit der gleichen Portnummer an den gleichen Server anbindet. Schickt dieser ebenfalls Daten, so wird die Kommunikation zwischen Server und Client gestört und die Ergebnisse können falsch werden.

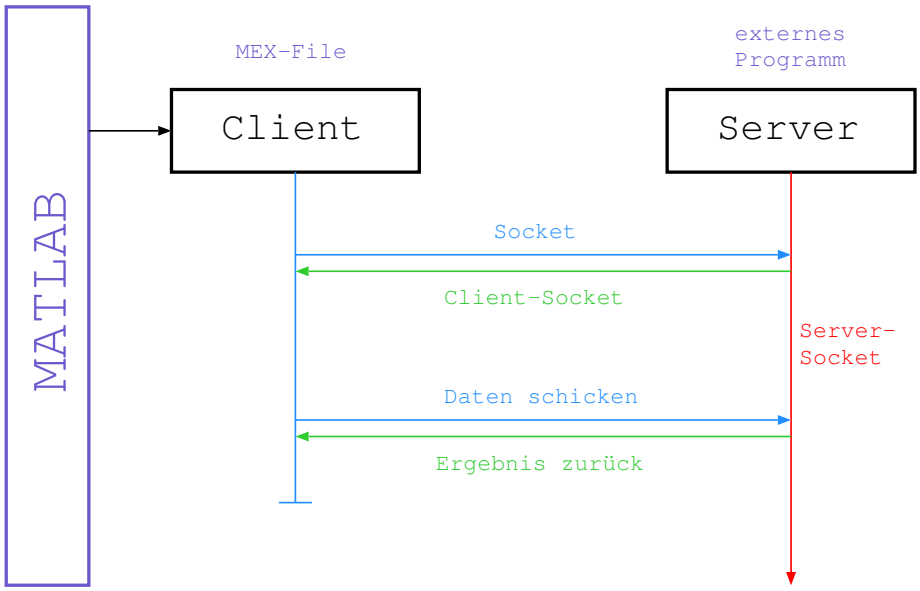

Abbildung 6.8: Kommunikation zwischen MATLAB und dem externen Programm

Bei dem entstandenen Programmpaket existiert nur ein **Client**, die Benutzeroberfläche. Der **Server** ist das externe Programm, das gleichzeitig mit dem Öffnen der Entwicklungsumgebung gestartet wird. Entscheidet sich der Benutzer, die Matrix extern zu speichern, so muss er verschiedene Funktionen zur Verfügung stellen.

MEX-Files für die verschiedenen Schnittstellen sind bereits implementiert und kompiliert und damit sind auch die Namen für die Funktionsaufrufe aus MATLAB festgelegt. Der Quellcode hierzu befindet sich im Verzeichnis **mexfiles**. Eine Kopie der kompilierten **MEX-Files** befinden sich bereits im gleichen Verzeichnis wie die Benutzeroberfläche.

Die Funktionsaufrufe unter MATLAB sehen folgendermaßen aus:

```
status = initialisiere(matrixbezeichnung,
         startvektorerzeugung, anzahl)
status = orthogonalisiere(I, flag_basis)
status = matrixvektor(tau, I, flag_basis,
         flag_matrix, flag_ergebnis, J)
matrix = skalarprodukt(I, flag_basis_I,
         J, flag_basis_J)
status = korrektur(tau, I, flag_basis,
         flag_matrix, flag_ergebnis, J)
status = linearkombination(I, flag_basis,
         koeffizienten, flag_ergebnis, J)
```

Abbildung 6.9: Funktionsaufrufe aus MATLAB

Die vom Benutzer zu schreibenden Funktionen haben dieselben Namen, erweitert um **_edit**. Die Übergabeparameter sind ebenfalls die gleichen. Zusätzlich werden allerdings noch Grössen von Feldern oder Zeichenketten und Zeiger auf die Speicherbereiche der extern gespeicherten Matrizen und deren Spaltenanzahlen übergeben. Genauere Erklärungen der Übergabeparameter finden sich bei den Erläuterungen der Funktionen, die vom Benutzer zu implementieren sind (Seite 53 bis 58).

Das Verzeichnis **schnittstellen** enthält die Dateien, die der Benutzer editieren muss. In diesen Dateien befinden sich bereits leere Funktionsrümpfe, da die Namen nicht mehr frei wählbar sind, ebenso wie die Übergabeparameter. Zusätzlich befindet sich hier noch ein Shellskript (**kompiliere.ksh**), mit dem die Dateien kompiliert werden können.

Standardmäßig werden die Dateien, die sich im Verzeichnis **schnittstellen** befinden, mit diesem Skript kompiliert (Aufruf: **kompiliere.ksh**).
Befindet sich die Funktion zum Orthogonalisieren nicht in der dafür vorgesehenen Datei **ortho.c**, so muss diese dem Skript mit der Option **-or Dateiname** angefügt werden (Aufruf: **kompiliere.ksh -or Dateiname**). Analog gilt dies für die anderen Funktionen. Als Option müssen immer die ersten beiden Buchstaben des Funktionsnamens angegeben werden.
Bei der Datei **func.h** handelt es sich um eine Header-Datei, die die benötigten Funktionen deklariert und die entsprechenden Bibliotheken einbindet. Die Dateien **kernel.c** und **schnittstellen.c** sind Dateien, die sich um die Kommunikation zwischen den Prozessen auf der Server- beziehungsweise Client-Seite kümmern. Diese müssen nicht weiter verändert werden.

6.3.2 Funktionsschnittstellen

Die Funktion ***initialisiere*** wird zu Beginn des Algorithmus aufgerufen. In dieser Funktion muss die Systemmatrix **A** bereitgestellt werden (Angabe eines Dateinamens, in der sich die Matrix befindet). Falls Startvektoren gewünscht werden, so muss zusätzlich noch eine weitere Datei angegeben werden, in der sich die Startvektoren befinden. Als dritter Parameter muss die Anzahl der bereitgestellten Startvektoren angegeben werden.
Der Rückgabeparameter gibt an, ob die Funktion erfolgreich war oder nicht. Dies ist bei allen Funktionen außer der Funktion ***skalarprodukt*** so.

Die vom Benutzer zu implementierende Funktion muss den Namen **initialisiere_edit** tragen. Unter MATLAB wird diese Funktion mit dem Namen **initialisiere** aufgerufen. Der bereits vorgegebene leere Funktionsrumpf befindet sich in der Datei **initia.c** und beinhaltet folgende Übergabeparameter:

status = initialisiere_edit(matrixbezeichnung, laenge_matrix, startvektorerzeugung, laenge_startvektoren, anzahl, matrix, groesse, V, spalten_V, W, spalten_W, Q, spalten_Q, help, spalten_help)		
matrixbezeichnung	char *	Dateiname, in der die Matrix steht
laenge_matrix	int	Länge des Strings **matrixbezeichnung**
startvektorerzeugung	char *	Dateiname, in der die Startvektoren stehen
laenge_startvektoren	int	Länge des Strings **startvektorerzeugung**
anzahl	int	Anzahl der Startvektoren
matrix	double ***	Zeiger auf den Speicherbereich der Matrix
groesse	int *	Zeiger auf die Größe der Matrix
V	double ***	Zeiger auf den Speicherbereich der Basis **V**
spalten_V	int *	Zeiger auf die Anzahl der Spalten von **V**
W	double ***	Zeiger auf den Speicherbereich der Basis **W**
spalten_W	int *	Zeiger auf die Anzahl der Spalten von **W**
Q	double ***	Zeiger auf den Speicherbereich der iterierten Vektoren
spalten_Q	int *	Zeiger auf die Anzahl der Spalten von **Q**
help	double ***	Zeiger auf den Speicherbereich der Hilfsvektoren
spalten_help	int *	Zeiger auf die Anzahl der Hilfsvektoren
status	double	1 erfolgreich 0 nicht erfolgreich

Tabelle 6.5: Übergabeparameter der Funktion initialisiere_edit

Die Übergabeparameter **matrix, groesse, V, spalten_V, W, spalten_W, Q, spalten_Q, help** und **spalten_help** bekommen alle Funktionen übergeben, damit sie wissen, wo sich die entsprechenden Speicherbereiche befinden, falls sie darauf zugreifen müssen. In der Funktion **initialisiere_edit** werden diese Zeiger mit **call-by-reference** übergeben, weil hier Speicher angelegt wird. In den anderen Funktionen werden sie mit **call-by-value** übergeben, da der Speicher bereits allokiert ist.

Die Funktion ***orthogonalisiere*** dient dazu, die bereitgestellten Startvektoren zu orthogonalisieren. Werden keine Startvektoren angegeben, so entfällt dieser Funktionsaufruf. Das Feld **I** enthält dabei die Indizes der zu orthogonalisierenden Vektoren. Das Flag gibt an, ob die Vektoren in die Basis aufgenommen werden oder nicht.

Der Benutzer muss die Funktion **orthogonalisiere_edit** in der Datei **ortho.c** implementieren, die als zusätzliche Parameter zum Aufruf aus MATLAB heraus noch die Grösse des Feldes **I**, Zeiger auf die Speicherbereiche der Matrix **A**, der Basis **V**, der Basis **W**, der Eigenvektornäherungen **Q** und der Hilfsvektoren sowie deren Spaltenanzahlen enthält:

status = orthogonalisiere_edit(I, size_I, flag_basis_I, matrix, groesse, V, spalten_V, W, spalten_W, Q, spalten_Q, help, spalten_help)		
I	int *	Indizes der gewünschten Vektoren
size_I	int	Grösse des Feldes **I**
flag_basis_I	int	1 für Vektoren der Basis **V** 2 für Vektoren der Basis **W** 3 für iterierte Vektoren
matrix	double **	Zeiger auf den Speicherbereich der Matrix
groesse	int	Größe der Matrix
V	double **	Zeiger auf den Speicherbereich der Basis **V**
spalten_V	int	Anzahl der Spalten von **V**
W	double **	Zeiger auf den Speicherbereich der Basis **W**
spalten_W	int	Anzahl der Spalten von **W**
Q	double **	Zeiger auf den Speicherbereich der iterierten Vektoren
spalten_Q	int	Anzahl der Spalten von **Q**
help	double **	Zeiger auf den Speicherbereich der Hilfsvektoren
spalten_help	int	Anzahl der Hilfsvektoren
status	double	1 erfolgreich 0 nicht erfolgreich

Tabelle 6.6: Übergabeparameter der Funktion orthogonalisiere_edit

Unter Fortran müssen alle Parameter, die von MATLAB an die externe Funktion übergeben werden und den Datentyp **int** haben, als Parameter mit Datentyp **double** (entspricht dem Fortran-Datentyp **double precision**) übergeben werden, da in Fortran der Datentyp **integer** (entspricht dem Datentyp **int** in C) nicht unterstützt wird.
Die Anzahl der Elemente eines Feldes oder einer Matrix können weiterhin als Parameter mit Datentyp **integer** übergeben werden, wenn sie nicht von MATLAB übergeben werden, sondern in der **Gateway-Routine** bestimmt werden.
Dies bedeutet bei den angegebenen Funktionen, dass die Flags mit Datentyp **double precision** entgegengenommen werden müssen. Die Größen der Indizevektoren beispielsweise können weiterhin mit **integer** übergeben werden, da ihre Größe nicht von MATLAB übergeben, sondern in der **Gateway-Routine** bestimmt wird.

Die Funktion ***matrixvektor*** berechnet das Matrix-Vektorprodukt. Der Parameter ***tau*** gibt den Shift an, ***I*** enthält die Indizes der gewünschten Vektoren. Die Flags geben an, ob es sich um Basisvektoren, iterierte Vektoren oder Hilfsvektoren, um die Systemmatrix, die Basis **V**, die Basis **W** oder die iterierten Vektoren handelt, ob mit der transponierten Matrix gerechnet werden soll und ob die neu erzeugten Vektoren zur Basis hinzugefügt werden sollen. Falls dies der Fall ist, so gibt ***J*** an, wo sie abgespeichert werden sollen.

Der Benutzer muss die Funktion **matrixvektor_edit** in der Datei **matvek.c** implementieren. Auch sie enthält als zusätzliche Parameter noch Zeiger auf die Speicherbereiche der Matrix **A**, der Basis **V**, der Basis **W**, der Eigenvektornäherungen **Q** und der Hilfsvektoren:

status = matrixvektor_edit(tau, I, size_I, flag_basis, flag_matrix, flag_ergebnis, J, flag_trans, matrix, groesse, V, spalten_V, W, spalten_W, Q, spalten_Q, help, spalten_help)		
tau	double	Bereichsangabe (Shift)
I	int *	Indizes der gewünschten Vektoren
size_I	int	Grösse des Feldes **I**
flag_basis	int	1 für Vektoren der Basis **V** 2 für Vektoren der Basis **W** 3 für iterierte Vektoren 4 für Hilfsvektoren 9 für von MATLAB übergebene Vektoren
flag_matrix	int	0 für Systemmatrix 1 für Basis **V** 2 für Basis **W** 3 für iterierte Vektoren
flag_ergebnis	int	1 für Hinzufügen zur Basis **V** 2 für Hinzufügen zur Basis **W** 3 für Hinzufügen zu den iterierten Vektoren 4 für Speichern als Hilfsvektor
J	int *	Indizes der Vektoren nachher
flag_trans	int	1 für transponierte Matrix 0 für nicht transponierte Matrix
matrix	double **	Zeiger auf den Speicherbereich der Matrix
groesse	int	Größe der Matrix
V	double **	Zeiger auf den Speicherbereich der Basis **V**
spalten_V	int	Anzahl der Spalten von **V**
W	double **	Zeiger auf den Speicherbereich der Basis **W**
spalten_W	int	Anzahl der Spalten von **W**
Q	double **	Zeiger auf den Speicherbereich der iterierten Vektoren
spalten_Q	int	Anzahl der Spalten von **Q**
help	double **	Zeiger auf den Speicherbereich der Hilfsvektoren
spalten_help	int	Anzahl der Hilfsvektoren
status	double	1 erfolgreich 0 nicht erfolgreich

Tabelle 6.7: Übergabeparameter der Funktion matrixvektor_edit

Die Funktion ***skalarprodukt*** berechnet das Skalarprodukt von Vektoren. Die Felder **I** und **J** geben die Indizes der Vektoren an und die Flags, ob die Vektoren sich unter den Basis- oder den iterierten Vektoren befinden. Der Rückgabewert **skal** enthält die berechneten Skalarprodukte.

Die Schnittstelle hierfür ist die Funktion **skalarprodukt_edit** in der Datei **skalar.c**:

skal = skalarprodukt_edit(I, size_I, flag_basis_I, J, flag_basis_J, matrix, groesse, V, spalten_V, W, spalten_W, Q, spalten_Q, help, spalten_help)		
I	int *	Indizes der gewünschten Vektoren
size_I	int	Grösse des Feldes **I**
flag_basis_I	int	1 für Vektoren der Basis **V** 2 für Vektoren der Basis **W** 3 für iterierte Vektoren 4 für Hilfsvektoren 9 für von MATLAB übergebene Vektoren
J	int *	Indizes der Vektoren
flag_basis_J	int	1 für Vektoren der Basis **V** 2 für Vektoren der Basis **W** 3 für iterierte Vektoren 4 für Hilfsvektoren 9 für von MATLAB übergebene Vektoren
matrix	double **	Zeiger auf den Speicherbereich der Matrix
groesse	int	Größe der Matrix
V	double **	Zeiger auf den Speicherbereich der Basis **V**
spalten_V	int	Anzahl der Spalten von **V**
W	double **	Zeiger auf den Speicherbereich der Basis **W**
spalten_W	int	Anzahl der Spalten von **W**
Q	double **	Zeiger auf den Speicherbereich der iterierten Vektoren
spalten_Q	int	Anzahl der Spalten von **Q**
help	double **	Zeiger auf den Speicherbereich der Hilfsvektoren
spalten_help	int	Anzahl der Hilfsvektoren
skal	double *	Skalarprodukt der Vektoren **I** und **J**

Tabelle 6.8: Übergabeparameter der Funktion skalarprodukt_edit

Die Funktion ***korrektur*** dient zur Lösung der Korrekturgleichung. Die Parameter sind identisch mit denen der Funktion ***matrixvektor***, außer dass das Flag **flag_trans** nicht benötigt wird.

Das gleiche gilt für die vom Benutzer zu schreibende Routine **korrektur_edit** in der Datei **korrekt.c**:

status = korrektur_edit(tau, I, size_I, flag_basis, flag_matrix, flag_ergebnis, J, matrix, groesse, V, spalten_V, W, spalten_W, Q, spalten_Q, help, spalten_help)		
tau	double	Bereichsangabe (Shift)
I	int *	Indizes der gewünschten Vektoren
size_I	int	Grösse des Feldes **I**
flag_basis	int	1 für Vektoren der Basis **V** 2 für Vektoren der Basis **W** 3 für iterierte Vektoren
flag_matrix	int	0 für Systemmatrix 1 für Vorkonditionierer
flag_ergebnis	int	0 für nicht Hinzufügen zur Basis 1 für Hinzufügen zur Basis
J	int *	Index, wo der neue Vektor hingeschrieben werden werden soll
matrix	double **	Zeiger auf den Speicherbereich der Matrix
groesse	int	Größe der Matrix
V	double **	Zeiger auf den Speicherbereich der Basis **V**
spalten_V	int	Anzahl der Spalten von **V**
W	double **	Zeiger auf den Speicherbereich der Basis **W**
spalten_W	int	Anzahl der Spalten von **W**
Q	double **	Zeiger auf den Speicherbereich der iterierten Vektoren
spalten_Q	int	Anzahl der Spalten von **Q**
help	double **	Zeiger auf den Speicherbereich der Hilfsvektoren
spalten_help	int	Anzahl der Hilfsvektoren
status	double	1 erfolgreich 0 nicht erfolgreich

Tabelle 6.9: Übergabeparameter der Funktion korrektur_edit

Die Funktion ***linearkombination*** dient zur Berechnung von Linearkombinationen von Vektoren. Das Feld **I** gibt hierbei wieder die Indizes der Vektoren an und das Flag kennzeichnet, ob es sich um Basis- oder iterierte Vektoren handelt. Das Feld **koeffizienten** enthält die Koeffizienten für die Linearkombination. Des Weiteren wird angegeben, wohin das Ergebnis gespeichert werden soll.

Auch hier existiert eine vorgefertigte Funktion **linearkombination_edit** in der Datei **linear.c**:

status = linearkombination_edit(I, size_I, flag_basis, koeffizienten, flag_ergebnis, J, matrix, groesse, V, spalten_V, W, spalten_W, Q, spalten_Q, help, spalten_help)		
I	int *	Indizes der gewünschten Vektoren
size_I	int	Grösse des Feldes **I**
flag_basis	int	1 für Vektoren der Basis **V** 2 für Vektoren der Basis **W** 3 für iterierte Vektoren 4 für Hilfsvektoren 9 für von MATLAB übergebene Vektoren
koeffizienten	double *	Koeffizienten für die Linearkombination
flag_ergebnis	int	1 für Hinzufügen zur Basis **V** 2 für Hinzufügen zur Basis **W** 3 für Hinzufügen zu den iterierten Vektoren 4 für Speichern als Hilfsvektor
J	int *	Index, wo das Ergebnis hingeschrieben werden soll
matrix	double **	Zeiger auf den Speicherbereich der Matrix
groesse	int	Größe der Matrix
V	double **	Zeiger auf den Speicherbereich der Basis **V**
spalten_V	int	Anzahl der Spalten von **V**
W	double **	Zeiger auf den Speicherbereich der Basis **W**
spalten_W	int	Anzahl der Spalten von **W**
Q	double **	Zeiger auf den Speicherbereich der iterierten Vektoren
spalten_Q	int	Anzahl der Spalten von **Q**
help	double **	Zeiger auf den Speicherbereich der Hilfsvektoren
spalten_help	int	Anzahl der Hilfsvektoren
status	double	1 erfolgreich 0 nicht erfolgreich

Tabelle 6.10: Übergabeparameter der Funktion linearkombination_edit

6.4 Formate

Es gibt drei Formate, die beim Einlesen erkannt werden:

1. binär
2. ASCII
3. Matrix-Market-Format.

Beim ASCII-Format handelt es sich um ein Format, das auf dem lateinischen Alphabet, wie es im modernen Englisch benutzt wird, basiert. ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) ist ein standardisierter Zeichensatz für Computer und andere Kommunikationseinrichtungen zur Textdarstellung. ASCII ist ein Code, der digital dargestellten Ganzzahlen in der normalen Schriftsprache geschriebene Zeichen zuordnet. Digitale Geräte können intern nur Zahlen verarbeiten, da sie Textinhalte als Zahlenfolgen senden, empfangen und verarbeiten. Daher ist ein solcher Code erforderlich (siehe auch [20]). Die Matrix muss dabei zeilenweise in der Datei abgespeichert sein.

Beim Binärformat ist die Matrix in einem internen Format abgespeichert, das nicht mit allgemein verfügbaren Programmen eingesehen werden kann. In der Regel ist es nicht zeilenbasiert und kann alle Zeichen des Zeichensatzes, also auch druckbare Steuerzeichen, enthalten. Es gibt spezielle Editoren, in denen Binärdateien gelesen und bearbeitet werden können.
Der Vorteil einer Binärdatei ist, dass sie meist schneller geladen, gelesen und gespeichert werden kann und auf Massenspeichern weniger Speicherplatz benötigt.
Ein Nachteil ist jedoch, dass diese Dateien nicht über Plattform- und Betriebssystemgrenzen hinweg austauschbar sind (siehe auch [20]).
In dem entstandenen Programmpaket ist es möglich, Matrizen im Binärformat abzuspeichern und einzulesen.

Das Matrix-Market-Format basiert auf dem ASCII-Format. Generell existieren drei verschiedene Formate:

- Matrix-Market-Exchange-Format,
- Harwell-Boeing-Exchange-Format,
- Coordinate-Text-File-Format,

wovon das letzte allerdings veraltet ist.

In der Arbeit wurden nur Dateien mit dem Matrix-Market-Exchange-Format getestet.

6.4.1 Matrix-Market-Exchange-Format

In diesem Format existieren zwei Unterformate:

- Coordinate-Format
- Array-Format

Das Coordinate-Format ist gebräuchlich für dünn besetzte Matrizen. Hier werden nur Nicht-Null-Elemente abgespeichert. Von jedem Nicht-Null-Element werden die Koordinaten explizit benannt.

Das Array-Format wird für dicht besetzte Matrizen benutzt. Hierbei werden alle Einträge in einer vordefinierten Ordnung (spaltenbasiert) aufgelistet.

Das Matrix-Market-Coordinate-Format sieht folgendermaßen aus:

```
%%MatrixMarket matrix coordinate real general
%
% comments
%
    M     N     L
    I_1   J_1   A(I_1,J_1)
    I_2   J_2   A(I_2,J_2)
    I_3   J_3   A(I_3,J_3)
          ...
    I_L   J_L   A(I_L,J_L)
```

Abbildung 6.10: Generelles Matrix-Market-Coordinate-Format

Die erste Zeile ist dabei die Header-Zeile, danach folgen Null oder mehr Kommentarzeilen, die mit % beginnen. Die erste Nicht-Kommentarzeile gibt, in dieser Reihenfolge, die Anzahl M der Zeilen, N der Spalten und L der Nicht-Null-Einträge an. Dahinter folgen L Zeilen (die L Einträge). Die erste Zahl gibt die Zeile, die zweite Zahl die Spalte und die dritte Zahl den Inhalt an. Die Header-Zeile gibt an, dass es sich um eine Matrix im Coordinate-Format handelt, dass die nachfolgenden numerischen Daten reell sind und die Matrix in der generellen Form dargestellt ist; das bedeutet, dass keine Symmetrie beim Abspeichern berücksichtigt wurde.

Hierzu ein Beispiel:

Folgende reelle 5×5 dünn besetzte Matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 6 & 0 \\ 0 & 10.5 & 0 & 0 & 0 \\ 0 & 0 & 0.015 & 0 & 0 \\ 0 & 250.5 & 0 & -280 & 33.32 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

wird im Matrix-Market-Coordinate-Format wie folgt dargestellt:

```
%%MatrixMarket matrix coordinate real general
5 5 8
1    1    1.000e+00
2    2    1.050e+01
3    3    1.500e-02
1    4    6.000e+00
4    2    2.505e+02
4    4   -2.800e+02
4    5    3.332e+01
5    5    1.200e+01
```

Abbildung 6.11: Matrix-Market-Coordinate-Format für eine reelle 5×5 Matrix

Es existieren ebenfalls Varianten des Coordinate-Formats für Matrizen mit komplexen oder ganzzahligen Einträgen. Außerdem gibt es Varianten, in denen die vorhandene Symmetrie einer Matrix berücksichtigt wird, um die Menge der Einträge signifikant zu verringern. Anstatt ***general*** findet man dann ***symmetric***, ***skew-symmetric*** (schief-symmetrisch) oder ***hermitian*** in der Header-Zeile. Da bei ***skew-symmetric*** die Diagonaleinträge Null sind, werden diese weggelassen. Bei symmetrischen Matrizen werden nur die Einträge des unteren Dreiecks aufgelistet.

6.4.2 Harwell-Boeing-Exchange-Format

Das Harwell-Boeing-Exchange-Format ist ein fixes Format für dünn besetzte Matrizen. Die Matrixeinträge sind dabei in einem 80 Spalten umfassenden Format fester Länge abgespeichert. Auch hier beginnt die Datei mit einem Header-Block, der mehrere Zeilen umfassen kann. Danach kommen zwei, drei oder vier Datenblöcke.
Der Header-Block enthält hierbei Informationen über das Speicherformat und die Speicherplatzanforderungen. Die Datenblöcke enthalten den Spaltenstartpointer, die Zeilenindizes und numerische Werte. Bei diesem Format können zusätzlich ***rechte Seiten*** für die Lösung eines Gleichungssystems angegeben werden. Sind keine ***rechten Seiten*** vorhanden, so enthält die Header-Datei vier Zeilen und zwei oder drei Datenblöcke, sonst fünf Zeilen und vier Datenblöcke.

Dieses Format ist in der entstandenen Arbeit bisher noch nicht berücksichtigt.

6.4.3 Coordinate-Text-File-Format

Das veraltete Coordinate-Text-File-Format ist ebenfalls für dünn besetzte Matrizen gedacht. Es ist wie folgt aufgebaut:

```
m     n     nz
i_1   j_1   val_1
i_2   j_2   val_2
      ...
i_nz  j_nz  val_nz
```

Abbildung 6.12: Coordinate-Text-File-Format

Die erste Zeile enthält drei Integerwerte, wobei der erste die Anzahl m der Zeilen, der zweite die Anzahl n der Spalten und der dritte die Anzahl nz der Nicht-Null-Einträge angibt. Danach folgen die Nicht-Null-Elemente, immer ein Eintrag pro Zeile, wobei zuerst der Zeilenindex i, dann der Spaltenindex j und zum Schluss der Wert $a(i, j)$ angegeben wird. Leerzeichen sind dabei nicht wichtig und Werte können sowohl in Gleitkommadarstellung (3.14) als auch in fixer Notation (3.14e00) angegeben werden.
Experimente zeigen, dass Daten im Coordinate-Text-File-Format ungefähr 30% größer sind als die entsprechenden Harwell-Boeing-Dateien.

Im Groben entspricht dies dem Coordinate-Format, nur dass beim Coordinate-Format noch eine Header-Zeile und Kommentarzeilen den eigentlichen Daten vorgeschaltet werden. In dieser Header-Zeile wird allerdings angegeben, ob die Matrix symmetrisch ist. Ist dies der Fall, muss nur etwa die Hälfte an Daten abgespeichert werden. Dies ist im Coordinate-Text-File-Format nicht möglich, da nirgendwo angegeben werden kann, ob Symmetrie vorliegt. Daher ist dieses Format für symmetrische Matrizen uneffektiv und wird in der entstandenen Arbeit nicht benutzt (siehe auch [19]).

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit ist eine Entwicklungsumgebung für iterative Eigenwertverfahren entstanden. Die Benutzeroberfläche ermöglicht ein effizientes Testen von Algorithmen mittels kleiner Matrizen. Ziel ist hierbei, herauszufinden, welche Algorithmen am besten und schnellsten für die jeweilige Problemstellung sind.

Es ist möglich, bestimmte Operationen, wie zum Beispiel das Lösen der Korrekturgleichung, mit eigenen, speziell optimierten Algorithmen außerhalb von MATLAB durchzuführen. Auch die Speicherung der Matrix und aller die Matrix betreffenden Informationen, wie zum Beispiel Basen und Eigenvektornäherungen, ist außerhalb von MATLAB möglich. Dann muss der Benutzer allerdings Routinen für die jeweiligen Matrixoperationen bereitstellen.

Besitzt der Benutzer keine eigenen externen Routinen zur Berechnung von Matrix-Vektorprodukt und Skalarprodukt, so hat er die Möglichkeit, die Matrix in MATLAB einzulesen. Hier kann er nun verschiedene Varianten des Jacobi-Davidson-Verfahrens testen oder aber die Lanczos- oder Arnoldi-Methode nutzen, um die gewünschten Eigenwerte und -vektoren zu berechnen. Besitzt er eine externe Funktion zur Lösung der Korrekturgleichung, so ist es möglich, nur diese Berechnung außerhalb von MATLAB durchzuführen.

Die externen Routinen müssen dabei nicht auf dem gleichen Rechner ausgeführt werden wie die Benutzeroberfläche, da mit **Sockets** gearbeitet wird, die eine Kommunikation über Rechnergrenzen hinweg ermöglichen. Eine Alternative zur Kommunikation über Sockets stellt die Bibliothek ***VISIT*** dar, die im Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich entwickelt wurde.
Bei der Kommunikation über Sockets muss der Server eine Portnummer angeben, mit der sich der Client verbinden muss. Diese Portnummer ist nicht exklusiv für die Kommunikation zwischen dem Server und dem Client reserviert. Es ist möglich, dass sich ein weiterer Client an den Server verbindet und Daten austauschen will. Damit wird die Kommunikation zwischen Server und Client gestört.
Dies wird bei der Kommunikation mittels ***VISIT*** unterbunden. Daher ist ***VISIT*** sicherer.

Das Tool befindet sich noch in der Entwicklungsphase und muss in Zukunft noch weiterentwickelt werden. Mögliche Erweiterungen sind zum Beispiel:

- Hinzufügen weiterer Algorithmen zur Berechnung der Eigenwerte und -vektoren
- Hinzufügen weiterer Formate für Matrizen, zum Beispiel Harwell-Boeing-Exchange-Format
- Berechnung von Eigenwerten innerhalb eines vorgegebenen Intervalls
- Implementierung weiterer möglicher Vorkonditionierer

- Implementierung weiterer Löser für die Korrekturgleichung
- Umstellung der Sockets auf ***VISIT***

Literaturverzeichnis

[1] Z. Bai, J. W. Demmel
On Swapping Diagonal Blocks in Real Schur Form
in Linear Algebra Application, 186 (1993), 73-95

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. van der Vorst
Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide
Philadelphia, Pa.: SIAM (2000)
ISBN : 0-89871-471-0

[3] E.R. Davidson
The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices
J. Comput. Phys. 17 (1975), 62-76

[4] A. Göke
Parallele Teilraumverfahren zur Bestimmung von Eigenwerten im Innern des Spektrums
Berichte des Forschungszentrum Jülich 3794 (2000)
ISSN: 0944-2952

[5] D.L. Harra II
On the Davidson and Jacobi-Davidson Methods for Large-Scale Eigenvalue Problems
Centre for Mathematics and its Applications, Australian National University, Canberra
AMS Number: 65F15

[6] C.G.J. Jacobi
Über ein leichtes Verfahren, die in der Theorie der Säkularströmungen vorkommenden Gleichungen numerisch zu lösen
Journal für die reine und angewandte Mathematik (1846), 51-94

[7] M. Jürgens
LaTeX- eine Einführung und ein bißchen mehr
Benutzerhandbuch (1995)
FZJ-ZAM-BHB-0134

[8] M. Jürgens
LaTeX- Fortgeschrittene Anwendungen oder: Neues von den Hobbits ...
Benutzerhandbuch (1995)
FZJ-ZAM-BHB-0135

[9] C. C. Paige. B. N. Parlett, H. A. van der Vorst
Approximate Solutions and Eigenvalue Bounds from Krylov Subspaces
Numer. Lin. Alg. Appl. 29 (1995), 115-134

[10] A. Quarteroni, R. Sacco, F. Saleri
Numerische Mathematik 1
Berlin: Springer-Verlag (2002)
ISBN : 3-540-67878-6

[11] M. Reißel
Three-dimensional eddy-current computation using Krylov subspace methods
in IMA Journal of Mathematics Applied in Business & Industry (1997) 8, 99-121

[12] D. Schräder
Eigenwertberechnung bei symmetrischen, positiv definiten Matrizen
Bachelorarbeit im Studiengang "Angewandte Informatik",
Georg-August-Universität Göttingen (2004)
ISSN: 1612-6793

[13] G. L. G. Sleijpen, H. A. van der Vorst
A Jacobi-Davidson iteration method for linear eigenvalue problems
SIAM Journal on Matrix Analysis and Applications, Volume 17, Number 2 (1996), 401-425

[14] B. Steffen
Subspace Methods for Sparse Eigenvalue Problems
in Modern Methods and Algorithms of Quantum Chemistry, Proceedings, Second Edition, Ed. J. Grotendorst, (2000), 307-314, NIC Series, Vol. 3
ISBN : 3-00-005834-6

[15] H. A. van der Vorst
Computational Methods for Large Eigenvalue Problems
P.G. Ciarlet and J.L. Lions (eds), Handbook of Numerical Analysis, Volume VIII, North-Holland (Elsevier), Amsterdam (2002)

[16] H. A. van der Vorst
Iterative Methods for Large Linear Systems
Mathematical Institute, Utrecht University, The Netherlands
Lecture notes on iterative methods (June 2000)

[17] D. S. Watkins
Fundamentals of Matrix Computations
John Wiley & Sons, Inc., New York, 2002 ISBN : 0-471-21394-2

[18] *Matlab Online-Dokumentation*
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml

[19] *Online-Dokumentation des Matrix Market Formats*
http://math.nist.gov/MatrixMarket/formats.html#MMformat

[20] *Online-Lexikon Wikipedia*
http://de.wikipedia.org

[21] *Online-Lexikon Mathworld*
http://mathworld.wolfram.com

Anhang A

Beispielmatrix

Die Matrix hat die Grösse 66×66, enthält 2211 Einträge und ist mittels des Matrix-Market-Exchange-Formats symmetrisch abgespeichert, d.h. nur die Einträge des unteren Dreiecks sind ausgelistet:

```
%%MatrixMarket matrix coordinate real symmetric
66 66     2211
 1  1  1.9903332860000e+00
 2  1  5.6791217991800e+02
 3  1  7.7578361440700e+02
 4  1 -1.3867966028800e+03
 5  1 -2.6785523152800e-01
 6  1 -4.6690425346000e-01
 7  1 -1.1304774307100e+00
 8  1  3.9205778992900e-01
 9  1  2.6219928853500e-02
10  1 -4.9028843549300e+02
 ⋮  ⋮          ⋮
63 63  3.9087765301100e+03
64 63  4.6377447261500e-01
65 63  4.6377447261500e-01
66 63 -8.7216412777700e+00
64 64  4.7132111749600e+03
65 64  1.4076095629800e+00
66 64 -1.4566358838000e-15
65 65  4.7132111749600e+03
66 65 -3.1481901065800e-15
66 66  1.3630769148600e+03
```